

Boosting Quantum Classifier Efficiency through Data Re-Uploading and Dual Cost Functions

Supplementary Documentation

Authors: Sara Aminpour^{1,2}, Mike Banad¹, Sarah Sharif^{1,2*}

Author Affiliations:

¹ School of Electrical and Computer Engineering, University of Oklahoma, Norman, OK 73019, USA

²Center for Quantum and Technology, University of Oklahoma, Norman, OK 73019 USA

*Corresponding author: Sarah Sharif (email: s.sh@ou.edu)

Table of contents

- Supplementary Note 1. Range of training samples and number of layers
- Supplementary Note 2. Evaluating LCP and non-LCP approaches for fidelity cost function in fixed and random datasets for 1-qubit classifier for four different minimization methods
- Supplementary Note 3: Evaluating LCP and non-LCP approaches for trace distance cost function in fixed and random datasets for 1-qubit classifier
- Supplementary Note 4: performance comparison of 5-layer single-qubit quantum classifiers using fidelity and trace distance cost functions across various classification tasks and dataset types
- Supplementary Note 5: Evaluating LCP and non-LCP approaches for fidelity in fixed and random datasets for 2-qubit and 2-qubit entangled classifiers
- Supplementary Note 6: Method
- Supplementary Note 7: Optimization Methods
- Supplementary Note 8: Comparing our developed code with original reference

Supplementary Note 1: Range of training samples and number of layers

Figure S1.1 illustrates the performance of a quantum classifier utilizing a fidelity cost function within a five-layer framework for circular pattern classification in a fixed dataset, employing the L-BFGS-B optimization method. The analysis encompasses training data up to 250 samples to benchmark our algorithm against the findings from the reference ¹. The diagram depicts training accuracy with a blue dashed line and test accuracy with a solid blue line, underscoring the algorithm's efficacy. A red dot highlights a notable benchmark from the reference, showing an 89% accuracy with 200 training samples, demonstrating parity with this published result. The inset provides a visual representation of the classification process. Notably, test accuracy begins at approximately 70%, rising impressively to 96% for a slightly expanded dataset of 210 samples. Remarkably, with as few as 60 training samples, the model achieves a test accuracy of 91.8%, and the discrepancy between training and test accuracy diminishes with the inclusion of 90 samples. This observation underscores the efficiency of our approach, highlighting its capability to reach high accuracy levels without necessitating extensive training data.

Figure S1.2 showcases a systematic evaluation of a circular pattern classification model across a spectrum of architectural depths, ranging from 1 to 5 layers. The graphical analysis reveals that models with a solitary layer lag in performance compared to those with increased layer counts, marking a clear trend: as the number of layers escalates, so does the model's classification accuracy. Specifically, a single-layer setup achieves a peak accuracy of 61.9%, whereas a more complex five-layer configuration significantly elevates this metric to 88.8%, even when limited to only 35 training samples. This observation underscores a critical insight—enhancing the model's depth systematically improves its predictive capabilities, a phenomenon consistent with the advantages afforded by the data reuploading strategy integral to our approach. Given this marked improvement in model efficacy with layer augmentation, the paper prioritizes an in-depth investigation and discourse on the five-layer model's architecture, focusing on its ability to optimize classification accuracy with efficient utilization of training data.

Supplementary Note 2: Evaluating non-linear and linear classification approaches for fidelity cost function in fixed and random datasets for 1-qubit classifier for four different minimization methods

Figure 2 illustrates a comparison of four distinct optimization techniques, namely L-BFGS-B, COBYLA, Nelder-Mead, and SLSQP, applied to the task of classifying the circle pattern. The comparison evaluates both training and test accuracies using a fixed dataset of 4000 test samples and 5 layers. Initially, all algorithms demonstrate a perfect training accuracy of 100% with just a single sample, a result that aligns with expectations. However, as we increase the sample size, a divergence in performance becomes evident for these four minimization methods. The L-BFGS-B method maintains a training accuracy close to 90%, showcasing its robustness against overfitting. In contrast, COBYLA, Nelder-Mead, and SLSQP show significant variability and a decline in training accuracy, indicating a susceptibility to overfitting. Interestingly, the peak accuracy for COBYLA, Nelder-Mead, and SLSQP is achieved with merely 50 samples, beyond which overfitting becomes a significant issue. This observation suggests that, unlike L-BFGS-B, which requires a minimum of 100 samples to achieve an accuracy of 92%, the other three methods can attain over 95% accuracy with only 50 samples. L-BFGS-B does not reach this high accuracy level at 100 samples, and its performance slightly declines with an increase in training samples after 150 training samples. This analysis highlights the critical importance of carefully selecting the number of training samples based on the minimization method used. The right choice can effectively prevent overfitting, thereby enhancing classification accuracy. This insight is crucial for optimizing machine learning models and ensuring their generalizability and efficiency in practical applications.

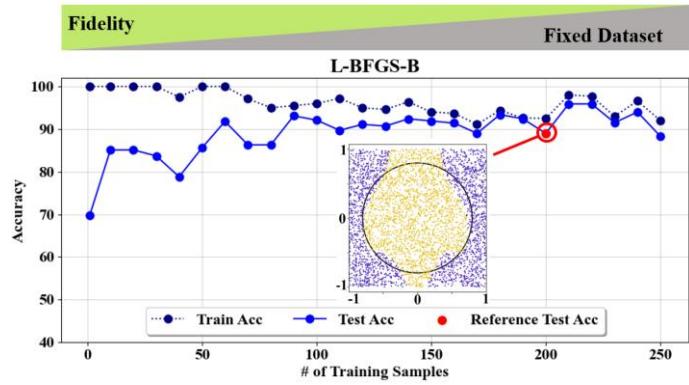


Figure S1.1 Train and test accuracy of fidelity for the 5-layer model of circle classification and fixed dataset for L-BFGS-B minimization method. The inset graph shows the visualization of a nonlinear classification reported on¹.

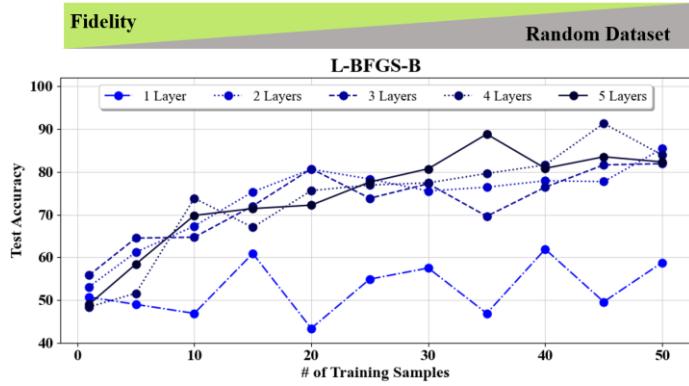


Figure S1.2. Evaluate the test accuracy of fidelity for circle classification and random dataset for L-BFGS-B minimization method, ranging from 1 to 5 layers.

Figure 3 delves into the accuracy of these four distinct minimization methods —L-BFGS-B, COBYLA, Nelder-Mead, and SLSQP— when applied to a fidelity cost function and a random dataset for circle classification. This analysis underscores a consistent trend across all methods: an initial increase in test accuracy corresponding to the rise in the number of training samples, yet fails to surpass a peak accuracy of 90%. This trend highlights the inherent challenges faced by these minimization methods when dealing with random datasets. In the L-BFGS-B method as depicted in figure 3(a), showcases a notable performance, achieving its highest test accuracy of 88.8% with 35 training samples. This point also marks the narrowest gap of 5% between training and test accuracy, indicating a relatively high level of model efficiency and generalization at this sample size. However, as the analysis progresses, it becomes apparent that increasing the number of training samples beyond this optimal point does not translate to improved performance. The gap between the train and test accuracy remains notably constant at around 10% even as the sample size is increased to 70 training samples. Transitioning to the COBYLA method, as depicted in figure 3(b), a different performance pattern emerges. Contrary to L-BFGS-B, COBYLA achieves its best test accuracy at 84.8% with a higher training sample equal to 70. This method experiences fluctuations, yet it is noteworthy that the gap between training and test accuracies exhibits a decreasing trend, suggesting a gradual improvement in model generalization compared to the initial stability seen with L-BFGS-B. Figure 3(c) focuses on the Nelder-Mead method, highlighting a decrease in the gap between training and test accuracies as the number of training samples increases, culminating in a maximum accuracy of 86.9% with 60 training samples. Figure 3(d) examines the SLSQP method, which shows an increase in test accuracy up to 50 training samples before demonstrating a decline in both training and test accuracies. This shows the SLSQP method is more prone to overfitting. The SLSQP method reaches a maximum accuracy of 86.7% when applied to a dataset of 50 samples. These results, as detailed in figure 6, provide vital insights into the performance of various minimization methods when working with a fidelity cost function and a random dataset. The diverse outcomes emphasize the importance of choosing an optimal number of training samples to prevent overfitting and enhance accuracy. This underlines the delicate balance needed to fully leverage these computational methods in practical scenarios.

Figure 4 illustrates a comparison of four different optimization techniques applied to the task of classifying line patterns, using fidelity-based cost function and the fixed dataset. The subplot (a) focuses on the performance of the L-BFGS-B method. Here, the training accuracy starts at a perfect 100% and impressively remains above 97% even as the number of training samples increases. Conversely, the test accuracy initiates at a relatively lower rate of 62.2% with just a single sample yet it progressively improves, reaching approximately 95% accuracy with 75 training samples and slightly declines for larger training samples. An initial notable gap between the training and test accuracy is evident, but this gap diminishes significantly as the dataset expands with more training data, indicating an improvement in the model's ability to generalize from the training to the unseen test data. The subplot (b) depicts the results obtained using the COBYLA algorithm, which exhibits a performance pattern similar to that of the L-BFGS-B method, consistently achieving 100% accuracy on the training data. The accuracy on the test set starts at 66.9% and steadily improves as more training samples are added, ultimately reaching 95% when 125 samples are used for training. The disparity between training and test set accuracies mirrors the pattern observed with the L-BFGS-B method, consistently manifesting across all training dataset sizes. The Nelder-Mead approach, shown in figure 4(c), achieves a notable test accuracy of 97.7% with 125 training samples. The inset provides a graphical visualization of line classification using this minimization method at this specific point, illustrating that the line classification performance is exceptionally well. The visualization clearly demonstrates the method's effectiveness in accurately separating the data points into distinct classes, highlighting the Nelder-Mead method's precision and robustness in handling line classification tasks with a substantial number of training samples. Furthermore, the training and test accuracy curves show a notably smaller gap, converging to the same value with training sets of 100 and 125 samples. The final subplot (d) evaluates the performance of the SLSQP method, which closely aligns with the results from the COBYLA method. The test set accuracy exhibits a progressive increase, rising from 62.7% to 96.6%. The disparity between the training and test accuracies is similar to that observed with the COBYLA method. In summary, all four optimization techniques demonstrate a reduction in overfitting as the training dataset size increases, ultimately achieving a test accuracy of at least 95% when training with 125 samples for this line classification task.

Figure 5 showcases an analysis of the classification accuracy obtained using the same minimization methods across random datasets. Consistently, a rise in the number of training samples correlates with an increase in test accuracy across all methods evaluated. Notably, with just 50 training samples, all methods surpass the 90% accuracy threshold. Specifically, in figure 5(a), the L-BFGS-B method reaches the peak accuracy of 92.8% with 50 training samples. It was observed that as the number of samples increased, the disparity between train and test accuracies for the L-BFGS-B method began to narrow, although this gap persisted in being slightly wider than that observed in the other methods. Figure 5(b) demonstrates that the COBYLA method, with the same number of samples, attains a superior accuracy of 93.5%. This suggests that COBYLA not only reaches high classification accuracy with a minimal dataset but also demonstrates better generalization compared to L-BFGS-B, as reflected by the narrower gap between its training and test accuracies. Figure 5(c) examines the Nelder-Mead method, showing its peak accuracy of 93% with 40 training samples, after which its accuracy slightly declines. Interestingly, the smallest disparity between training and test accuracies—only 1.8%—occurs in 50 training samples. Despite slightly lower accuracy at this point, this smallest gap signifies that the Nelder-Mead method achieves a remarkable balance between learning from the training data and generalizing to unseen data, highlighting its efficiency and potential for precise model tuning. Figure 5(d) illustrates that the SLSQP method achieves an impressive peak test accuracy of 96.4% for line classification using a random dataset, attained with 45 training samples. At this juncture, the discrepancy between training and test accuracies is notably small, indicating a high level of model precision and generalization. Like the Nelder-Mead method, the SLSQP method exhibits a nonmonotonic increment in test accuracy as a function of training samples, as indicated by the irregular slope of test accuracy. This fluctuation suggests that for these methods, adding more training samples does not straightforwardly translate to higher test accuracies, highlighting the complexity of optimizing model performance across different minimization techniques.

A comparison of figures 2 and 4 reveals that the accuracy curves for line classification are more stable and consistent across all optimization techniques when compared to those for circle classification. The accuracy values for classifying circle patterns display greater variability and fluctuations than those observed in the line classification task. The observed differences in performance between circle and line classification could stem from several technical factors: (1) Line classification likely represents a more straightforward pattern that aligns better with the linear decision boundaries most classifiers are adept at identifying. In contrast, circle classification involves recognizing more complex, non-LCP, which can challenge the classifiers' ability to generalize from the training data without overfitting or underfitting. (2) The algorithms applied for circle classification might be more prone to getting trapped in local minima due to the more intricate decision boundaries required to accurately classify circular patterns. This can hinder the optimization process, leading to increased fluctuations in classification accuracy as the model struggles to find the global optimum. (3) The differences in performance may also reflect the inherent adaptability of the algorithms to the specific types of classification tasks with the geometric properties. A comparative analysis of Figures 6 and 8 indicates that the specific characteristics of the classification problem significantly affect the potential to attain higher accuracy with fewer samples. The fluctuations in the line classification pattern are less pronounced than those in the circle classification pattern. This observation underscores the importance of selecting appropriate optimization methods based on the complexity of the classification problem.

Supplementary Note 3: Evaluating non-linear and linear classification approaches for trace distance cost function in fixed and random datasets for 1-qubit classifier

Figure 6 showcases the effectiveness of the trace distance cost function in classifying circular patterns within a fixed dataset. In subplot (a), the L-BFGS-B minimization method achieves its highest test accuracy at 79.2% with a dataset comprising 100 training samples. Subplot (b) examines the performance of the COBYLA method, which displays greater variability in training accuracy than L-BFGS-B but ultimately achieves a higher peak test accuracy of 84.6%, also with 100 training samples. Notably, COBYLA demonstrates enhanced generalization capabilities relative to other methods, as indicated by the narrower margin between its training and testing accuracies. This performance suggests that, when applied alongside the trace distance cost function, the COBYLA method is particularly adept at optimizing parameters for improved generalization to unseen testing data. An accompanying visualization within the inset illustrates the classification of circular patterns at this accuracy peak. In subplot (c), the analysis shifts to the performance of the Nelder-Mead method, which records its optimal test accuracy at 72.6% utilizing 60 training samples. This method exhibits signs

of overfitting, a condition where the model learns the training data too closely and fails to generalize well to new, unseen data. Despite a narrowing gap between training and testing accuracies as the number of training samples grows, a concurrent decline in training accuracy is observed, which adversely affects the overall test accuracy. This pattern suggests a limitation in the Nelder-Mead method's capacity to effectively handle the trace distance cost function, likely due to its inherent characteristics such as reliance on simplex-based optimization, which might struggle with the complexity of the trace distance landscape. Consequently, this method appears less suited for tasks requiring robust generalization from the trace distance cost function, particularly in scenarios demanding accurate classification of complex patterns with a limited dataset. In subplot (d), the focus turns to the SLSQP method which attains its peak test accuracy at 83.6% with a dataset of 100 training samples. The disparity between training and testing accuracy contracts by increasing the training samples, indicating an improvement in the model's ability to generalize from the training to the testing dataset. However, even at the point of 100 training samples, the gap between training and testing accuracies, while reduced, remains significant. This persistent gap suggests that while the SLSQP method is effective at learning and generalizing from the given data, there is still a margin for optimization to further bridge the difference in accuracies. Each optimization technique successfully minimizes the cost function and attains perfect accuracy on the training set using a comparatively small number of samples. However, their performance varies considerably when it comes to generalizing to the test set. This highlights the crucial role played by the choice of optimization algorithm in determining the overall effectiveness of the model. In conclusion, when considering the fixed dataset and the trace distance cost function, the COBYLA method demonstrates superior performance in optimizing the parameters to generalize effectively to unseen test data. Compared to the other techniques evaluated, it necessitates fewer training samples to achieve satisfactory accuracy on the test set.

Figure 7 illustrates how the accuracy on both the training and test sets evolves as the number of training samples grows, specifically for the task of classifying circular patterns using the trace distance cost function, evaluated on a randomly generated dataset. Similar to all scenarios analyzed so far, a common pattern emerges where test accuracy begins at a relatively low level for all minimization methods but demonstrates a consistent increase as more training data is provided. This trend highlights the methods' capacity to effectively learn distinguishing features, thereby enhancing their ability to generalize to unseen data. Specifically, in subplot (a), the L-BFGS-B method illustrates impressive learning efficiency, with test accuracy exceeding 70% after incorporating just 40 training samples and achieving its highest test accuracy of 77.8% with 45 training samples. In subplot (b), the COBYLA method's performance is slightly lower compared to L-BFGS-B, plateauing at a test accuracy of 72.8% with 45 training samples. This performance indicates that while COBYLA may be susceptible to some degree of overfitting, it nonetheless achieves a reasonable level of generalization. Subplot (c) explores the Nelder-Mead method, which reaches its peak test accuracy of 75.1% with 50 training samples. Subplot (d) utilizes the SLSQP method, which shows fluctuations in its training accuracy remaining above 80%. The test accuracy for SLSQP was enhanced significantly, reaching 74.6% with 50 samples. This fluctuation and eventual rise in test accuracy underscores the method's potential for optimizing classification tasks, despite the initial variability. In sum, the L-BFGS-B method stands out for achieving the highest test accuracy among the methods evaluated, requiring only 45 training samples to reach this optimum on a random dataset. Summarily, employing the trace distance cost function across these various minimization strategies yields test accuracy ranging from 65% to 78% on the random dataset, illustrating the function's effectiveness and the distinct performance capabilities of each minimization method.

Figure 8 offers a comparative analysis of the accuracy achieved by four different optimization methods when applied to a trace distance cost function for line pattern classification using a fixed dataset. Subplot (a) highlights the L-BFGS-B method, showcasing its high level of stability in training accuracy. The test accuracy shows a steady increase, reaching 91.8% with 100 training samples. While there is a substantial gap between the accuracies of the training and test sets at the outset, this difference gradually narrows as more training samples are introduced. This highlights the L-BFGS-B method's capacity to adapt and learn more complex patterns effectively, demonstrating robustness and in leveraging larger datasets for improved generalization. The subplot (b) illustrates the results obtained using the COBYLA method. In contrast to the L-BFGS-B approach, the accuracy of the training set shows greater fluctuations, even experiencing a drop to 56.9% at one instance before rebounding. The test accuracy follows a similar pattern to that seen in L-BFGS-B,

beginning at 49.8% and increasing to 87.4%. Once the training set size reaches 80 samples, both the training and test accuracies seem to reach a plateau, slightly below the 90% mark. In subplot (c), the Nelder-Mead method starts with a modest test accuracy of 55.3%, which significantly improves to 87% with the addition of 60 training samples demonstrating a similar trend as the L-BFGS-B method. Initially, a pronounced gap exists between training and test accuracies, which persists until the dataset is expanded to include 80 training samples. Beyond this point, the sign of overfitting emerges, as demonstrated by a decline in training accuracy while test accuracy plateaus. For 100 training samples, the test accuracy interestingly becomes 2% higher than the training accuracy, indicating a unique inversion where the model performs slightly better on unseen data than on the training set itself, a rare occurrence that may suggest the model has reached a point of optimization where it generalizes exceptionally well to new data. The subplot (d) of figure 11 presents the results of the SLSQP method. Notably, this technique achieves the highest accuracy on the test set, reaching 93.3% using just 40 training examples. The SLSQP method appears to be the most appropriate choice for trace distance classification tasks, as it exhibits a smaller discrepancy between its performance on the training and test datasets. The inset provides a visual representation of the SLSQP's performance at this specific point. To summarize, all optimization methods demonstrate an upward trajectory in test accuracy as the size of the training dataset increases, suggesting enhanced generalization capabilities of the model. Among the four techniques evaluated, the SLSQP method seems to strike the most favorable balance between its performance on the training and test sets.

Figure 9 presents a comparison of different optimization techniques when applied to the task of classifying line pattern using a randomly generated dataset and a cost function based on trace distance. In subplot (a), we examine the performance of the L-BFGS-B method, which attains its peak test accuracy of 86.3% with 55 training samples. Before reaching this point, the method's test accuracy demonstrated considerable variability, oscillating between 70% and 80% as the number of training samples ranged from 20 to 50. However, a notable improvement occurs when the dataset is expanded to 55 training samples, at which the test accuracy leaps to 86.3%, effectively surpassing the earlier fluctuation band. This pivotal moment also marks the occurrence of the smallest gap between training and test accuracies, showcasing a significant enhancement in the model's ability to generalize from the training dataset to unseen data, thereby achieving an optimal balance at this specific training sample size. Subplot (b) delves into the efficacy of the COBYLA optimization method, which achieves its highest test accuracy of 86.8% with a relatively smaller dataset of 35 training samples. Beyond this optimal threshold, signs of overfitting become apparent, as both training and test accuracies start to decline. This pattern suggests that while the COBYLA method is highly effective up to a certain point, adding more training samples beyond this number paradoxically hampers the model's performance. The decline in accuracy indicates that the model begins to memorize the training data rather than learning to generalize, leading to a decrease in its ability to accurately predict outcomes on unseen data. This observation underscores the importance of identifying the ideal number of training samples to maximize the effectiveness of the COBYLA method without crossing into the territory of overfitting. In subplot (c), the focus is on the Nelder-Mead optimization method, which shows some fluctuations in performance before reaching its maximum test accuracy. It successfully achieves a test accuracy of 88.1% with 40 training samples. However, akin to the pattern observed with the COBYLA method, the Nelder-Mead method also sees a decline in both training and test accuracies when additional training samples are added beyond this optimal number. This decline serves as a clear indication of the onset of overfitting, suggesting that while the Nelder-Mead method can efficiently utilize a certain number of training samples to improve its predictive accuracy, exceeding this number leads to a reduction in model performance. In subplot (d), a more continuous and stable increase in test accuracy is observed with each increase in the number of training samples. This trend results in the highest test accuracy being recorded at 88.3% with 55 training samples. Unlike the previous methods discussed, this subplot suggests a method that maintains its efficiency and ability to generalize well without showing immediate signs of overfitting up to this point. The gradual and consistent improvement in test accuracy highlights the method's effective learning curve and suggests an optimal balance between learning from the training data and applying this knowledge to unseen data.

Supplementary Note 4: performance comparison of 5-Layer single-qubit quantum classifiers using fidelity and trace distance cost functions across various classification tasks and dataset types

Figure S4.1 offers a comparative analysis of the highest accuracies achieved for two distinct classification patterns – linear (line) and non-linear (circle) – across the four distinct minimization methods when applied to both random and fixed datasets within the context of a fidelity cost function. The analysis reveals a notable trend: in circle classification tasks, the fixed dataset consistently yields higher accuracies than their random counterparts for all tested minimization methods. This suggests that the inherent geometric complexities of non-LCP may align more closely with the simpler structure of fixed datasets, thereby facilitating more accurate classification. Similarly, for line classification, the fixed dataset leads to enhanced accuracies with the L-BFGS-B and SLSQP methods, indicating these methods' effectiveness in leveraging structured data to accurately discern linear relationships. However, the random dataset achieves better accuracy when classified using the Nelder-Mead method. This could suggest that the Nelder-Mead method, known for its simplicity and direct search approach, might be particularly adept at navigating the stochastic nature of random datasets to identify linear patterns. Across all algorithms, the task of classifying non-LCP, especially within random datasets, emerges as inherently challenging. This complexity likely stems from the algorithms' varying abilities to parse and learn from the unpredictable variance found in random datasets, as well as the added difficulty of accurately modeling non-linear relationships. The findings underscore the critical importance of selecting the appropriate minimization method based on the dataset's nature and the classification task's geometric complexity to optimize classification accuracy.

Figure S4.2 provides the performance comparison of two distinct classification patterns—line and circle—across four different minimization methods when applied to both random and fixed datasets, this time employing the trace distance cost function. A pivotal observation emerges when comparing the performance of circle classification with a fixed dataset (circle/fixed) against the fidelity cost function results presented in figure S4.1. It is evident that the accuracies achieved using the trace distance cost function are notably lower across all minimization methods compared to those obtained with the fidelity cost function. This discrepancy highlights the inherent challenges and differences in how each cost function interacts with the underlying data and the classification task at hand. The trace distance cost function, known for quantifying the distinguishability between quantum states, may present a more complex landscape for optimization, particularly when applied to classical data patterns such as lines and circles. This complexity could lead to lower classification accuracy as the minimization methods struggle to navigate the nuances of the trace distance landscape effectively. Such an observation underscores the importance of cost function selection in machine learning tasks, emphasizing that the

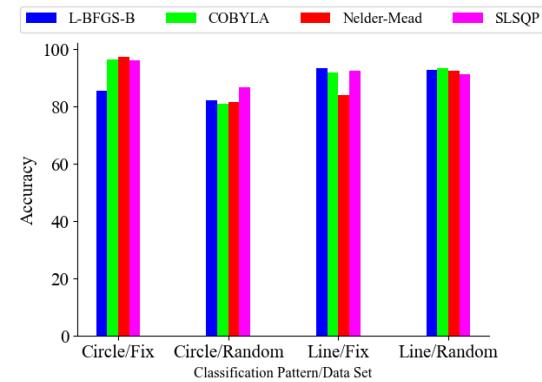


Figure S4.1. Evaluating of Fidelity cost function test accuracy of 5-layer model across 50 samples for LCP and non-LCP problems for random and fixed datasets in four minimization methods.

Figure S4.2. Evaluating of trace distance test accuracy of 5-layer model across 50 samples for LCP and non-LCP problems for random and fixed datasets in four minimization methods.

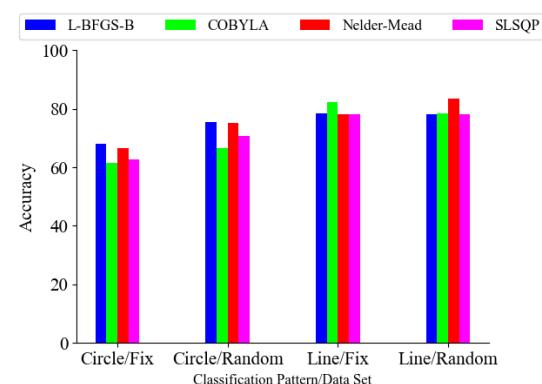


Figure S4.2. Evaluating of trace distance test accuracy of 5-layer model across 50 samples for LCP and non-LCP problems for random and fixed datasets in four minimization methods.

choice of cost function can significantly impact the model's ability to learn and generalize from the data. The comparative analysis in figure S4.2 serves as a testament to the nuanced interplay between cost functions, dataset types (fixed vs. random), and the geometric nature of the classification patterns, offering valuable insights into optimizing classification accuracy through strategic method and cost function selection.

In addition, the fixed dataset achieves superior accuracy specifically when employing the COBYLA minimization method, indicating a unique synergy between COBYLA's optimization strategy and the structured nature of fixed datasets for LCP. Conversely, for the random dataset, there's a notable trend where it consistently outperforms the fixed dataset across all other minimization methods, suggesting that the stochastic characteristics of random datasets may be better suited to the optimization landscapes these methods navigate, particularly for LCP. In circle classification tasks, the random dataset not only demonstrates improved accuracy over the fixed dataset for all minimization methods but also reinforces the observation that random datasets generally offer a more favorable context for the trace distance cost function across both classification patterns. This enhancement in accuracy with random datasets could be attributed to the trace distance cost function's sensitivity to the variances within the dataset, allowing for more effective differentiation and classification of non-LCP like circles when the data is less predictable.

Supplementary Note 5: Evaluating non-linear and linear classification approaches for fidelity in fixed and random datasets for 2-qubit and 2-qubit entangled classifiers

Focusing on figure 10(a), we observe the performance of a single-qubit system applied to a LCP pattern. The system demonstrates a steep initial learning curve, with accuracy rapidly increasing from 51.6% to 92% after just 75 training samples. This sharp rise highlights the single-qubit system's ability to efficiently learn and generalize from a relatively small dataset. The notable jump in accuracy suggests that a properly trained single-qubit classifier can capture the essential features of the LCP task with high precision. After reaching 92% accuracy at 75 training samples, the system stabilizes, maintaining a test accuracy consistently in the range of 92% to 97.7% as the training sample size increases to 125. The minimal fluctuation in accuracy indicates a robust performance, with the single-qubit system effectively avoiding overfitting even as the training data expands. The stable test accuracy underscores the system's reliability and suitability for LCP tasks where computational simplicity and consistent performance are crucial. In terms of computational cost, as shown in figure 1(d), the single-qubit system exhibits a gentle increase in computational time, reaching 62.15 seconds for 250 training samples. This computational efficiency, coupled with the system's stable accuracy, makes the single-qubit classifier an appealing option for linear problems, particularly in scenarios where computational resources are limited but high accuracy is still required.

In figure 10(b), the performance of the 2-qubit classifier in a LCP task shows a more gradual improvement in accuracy compared to the single-qubit system. The initial accuracy is relatively high, starting at 73.2% with just one training sample, which suggests that the additional qubit provides a more robust representation of the problem space even with minimal training. As the number of training samples increases to 75, the accuracy rises steadily, reaching 94.1%. This gradual improvement, as opposed to the sharp jump seen in the single-qubit system, highlights the ability of the 2-qubit classifier to build on its already strong initial performance with increasing training data. Beyond 50 training samples, the 2-qubit classifier continues to demonstrate incremental gains, eventually peaking at around 95.7% test accuracy with 175 training samples. Notably, the test accuracy fluctuates between 92% and 96% throughout this range, suggesting that while the system performs consistently well, there are slight variations in how the test data is classified as more training samples are introduced. These fluctuations could indicate that the system is sensitive to the nature of the training data or potentially approaching the limits of its capacity for linear classification. From a computational perspective, shown in Figure 1(e), the 2-qubit classifier exhibits a significant increase in computational time as the number of training samples grows. By the time the training sample size reaches 250, the computational time extends to around 260 seconds. This is a sharp contrast to the single-qubit system, illustrating the tradeoff between the enhanced accuracy and robustness offered by the 2-qubit classifier and the increased computational demands. For LCP tasks, this suggests that while the 2-qubit classifier provides higher initial accuracy and steady performance improvements, it comes at the cost of a much higher computational burden, making it potentially less suitable for scenarios where time or resources are constrained.

Examining figure 10(c), the performance of the 2-qubit entangled classifier in a LCP task reveals a distinctive pattern when compared to non-entangled systems. The initial accuracy is relatively low, starting at 51.3% with just one training sample. This suggests that the entanglement introduces complexities that make the system less effective in identifying patterns from very limited data. However, as the number of training samples increases to 75, the system exhibits a steep improvement in accuracy, reaching 93.3%. This rapid climb indicates that while the entangled system may struggle with very small datasets, it quickly capitalizes on additional training samples to enhance its classification performance. As the training samples continue to increase beyond 75, the 2-qubit entangled classifier shows notable fluctuations in accuracy, ranging between 88% and 97.5%. These fluctuations, which are more pronounced than those seen in the single-qubit or non-entangled 2-qubit classifier, suggest that entanglement introduces both benefits and challenges. On one hand, the system achieves the highest peak accuracy (97.5%) among all three systems, demonstrating its potential for superior performance. On the other hand, the variability in test accuracy highlights the sensitivity of the entangled system to the training data, possibly indicating overfitting or instability when processing larger datasets. In terms of computational cost, as shown in figure 10(f), the 2-qubit entangled classifier mirrors the trend seen in the non-entangled 2-qubit classifier, with computational time increasing significantly as the number of training samples rises. At 250 training samples, the computational time reaches 260 seconds, similar to the non-entangled classifier. Despite this computational burden, the 2-qubit entangled classifier offers a potential advantage in terms of peak accuracy, making it a compelling choice for applications where achieving the highest possible accuracy is paramount, even if it comes with the tradeoff of greater computational complexity and variability in performance.

In comparing the classifier, we observe clear tradeoffs between simplicity, stability, and computational complexity. The single-qubit classifier is the most stable and computationally efficient but may not reach the same peak accuracies as the more complex systems. The 2-qubit classifier offers higher initial accuracy and consistent improvement but requires significantly more computational resources. Finally, the 2-qubit entangled system, while achieving the highest peak accuracy, also introduces greater instability and computational demands, making it best suited for scenarios where peak performance is the priority, and computational cost is less of a concern. Ultimately, the choice of system depends on the specific requirements of the classification task, such as whether stability, computational efficiency, or peak accuracy is the primary objective.

Figure 11 presents a comprehensive analysis of two quantum classifiers - a 2-qubit classifier and a 2-qubit entangled classifier for non-LCP. The results are displayed across six subplots, labeled (a) through (f), which provide insights into the performance and characteristics of these classifiers under various conditions. Subplots (a) and (b) show the train and test accuracies as a function of the number of training samples for the 2-qubit and the 2-qubit entangled classifiers, respectively. In both cases, we observe that the accuracies generally improve as the number of training samples increases. However, the 2-qubit classifier (a) shows higher initial test accuracy, 73.5%, and a more stable performance across different sample sizes. The 2-qubit entangled classifier (b) starts with lower test accuracy, 47.6% but shows significant improvement as the sample size increases. Both classifiers seem to converge in terms of train and test accuracy around 175 training samples, which explains why this number was chosen for subsequent analyses. Subplots (c) and (d) illustrate how the number of layers in the quantum circuit affects the accuracies of the classifiers for a specific number of training samples. For the 2-qubit classifier (c), we see a general upward trend in both train and test accuracies as the number of layers increases, with some fluctuations. The 2-qubit entangled classifier (d) shows a more pronounced improvement with increasing layers, especially in the early stages. Both classifiers appear to reach a plateau in performance after about 12-15 layers, suggesting that further increases in circuit depth may not yield significant improvements. Subplots (e) and (f) depict the computational time required as the number of layers increases for the 2-qubit and the 2-qubit entangled classifiers, respectively. Both show a clear exponential growth in computational time as the number of layers increases. This trend is consistent across both classifiers, indicating that the computational cost scales similarly regardless of whether entanglement is used. Comparing the classifiers overall, we can see that the 2-qubit classifier generally achieves higher accuracies with fewer training samples and maintains more consistent performance across different numbers of layers. The 2-qubit entangled classifier, while starting with lower accuracy, shows more dramatic improvements as both the number of training samples and layers increase. This

suggests that entanglement might provide additional expressive power to the classifier, allowing it to capture more complex patterns in the data as the circuit depth increases. However, this potential advantage comes at the cost of increased sensitivity to the number of training samples and layers, as evidenced by the more volatile accuracy curves in subplots (b) and (d). The computational time plots (e) and (f) remind us that increasing the number of layers quickly becomes computationally expensive for both classifiers, which is an important consideration in practical applications. In conclusion, these results provide valuable insights into the trade-offs between accuracy, circuit complexity, and computational cost for quantum classifiers, highlighting the potential benefits and challenges of using entanglement in quantum machine learning tasks.

Figure 12 presents a comparative analysis of four optimization algorithms (COBYLA, L-BFGS-B, NELDER MEAD, and SLSQP) applied to a LCP using a quantum circuit with 2 qubits. The experiment uses a random dataset with 250 training samples and employs a fidelity cost function to measure the performance. The figure includes subplots depicting accuracy and computational time for both 2-qubit and 2-qubit entangled classifiers. In terms of accuracy, both training and test accuracies are generally high across all algorithms. However, there are subtle differences between the algorithms. As shown in figure 12(a), for the 2-qubit entangled classifier, the average test accuracy is approximately 2% higher than the 2-qubit non-entangled classifier. In terms of individual performance, the L-BFGS-B minimization method consistently achieves the highest test accuracy, reaching 96.3% for non-entangled and 97% for entangled classifiers. The overall variation in test accuracy between the highest and lowest performing algorithms is 2.3%. For 2-qubit non-entangled classifier, COBYLA exhibits the lowest test accuracy at 94%, while for 2-qubit entangled classifier, NELDER MEAD achieves the lowest test accuracy of 95.3%. Computational time analysis reveals interesting patterns across both classifiers. In figure 12(c) the 2-qubit classifier, computational time varies widely from 9 to 90 minutes. COBYLA stands out as the fastest method, completing the task in just 9 minutes, while L-BFGS-B and NELDER_MEAD are the most time-consuming at 90 and 89 minutes respectively. SLSQP occupies a middle ground, requiring 45 minutes. In figure 12(d) the 2-qubit entangled classifier generally shows improved computational efficiency. While COBYLA maintains its swift performance at 9 minutes, other methods see reduced execution times. Most notably, L-BFGS-B improves from 90 to 71 minutes, a significant reduction, while NELDER_MEAD and SLSQP methods remain at 87 and 44 minutes respectively. In conclusion, this analysis reveals that the 2-qubit entangled classifier generally outperforms the 2-qubit non-entangled classifier in both accuracy and computational efficiency. The L-BFGS-B method consistently provides the highest accuracy, albeit at a higher computational cost. COBYLA emerges as a well-balanced option, offering good accuracy with minimal computational time, particularly in the 2-qubit entangled classifier. These findings underscore the significant impact of minimization method selection on both accuracy and computational time in quantum machine learning tasks. Furthermore, the 2-qubit entangled classifier's closer alignment of train and test accuracies suggests enhanced generalization capabilities, a crucial factor in practical machine learning applications.

Figure 13 shows a comprehensive comparison of different optimization methods for non-LCP using both 2-qubit and 2-qubit entangled classifiers for a specific random dataset. This analysis encompasses four optimization techniques: COBYLA, L-BFGS-B, NELDER_MEAD, and SLSQP, evaluating their performance based on accuracy and computational time for 250 number of training samples. In the accuracy graphs (a) and (b), we observe distinct performance patterns between the 2-qubit and 2-qubit entangled classifiers. For the 2-qubit classifier, L-BFGS-B demonstrates the highest accuracy, with both train and test accuracies exceeding 90%. COBYLA shows the lowest performance, with a test accuracy of 76.7% and train accuracy 81.4%. NELDER_MEAD and SLSQP exhibit intermediate performance, with test accuracies in the 82-87% range. The 2-qubit entangled classifier, depicted in graph (b), shows overall improved accuracy across all methods. L-BFGS-B maintains its superior performance, while COBYLA shows significant improvement, reaching accuracies to 85.4%. Notably, the gap between train and test accuracies is generally smaller in the 2-qubit entangled classifier, suggesting better generalization. The computational time graphs (c) and (d) reveal interesting efficiency patterns. In the 2-qubit classifier, COBYLA is the fastest method, requiring only 9 minutes. L-BFGS-B, despite its high accuracy, is the most time-consuming at 130 minutes. NELDER_MEAD takes 89 minutes, while SLSQP requires 45 minutes. The 2-qubit entangled classifier (graph d)

shows generally reduced computational times. COBYLA remains the fastest, maintaining its 9-minute runtime. L-BFGS-B shows the most dramatic improvement, reducing its time to 81 minutes. Interestingly, NELDER_MEAD in the 2-qubit entangled classifier takes slightly longer than L-BFGS-B, at 88 minutes. SLSQP maintains a consistent performance of about 42 minutes in both systems. These results highlight the trade-offs between accuracy and computational efficiency in quantum machine learning tasks. The 2-qubit entangled classifier demonstrates superior performance in both accuracy and computational time across all methods. L-BFGS-B consistently provides the highest accuracy but at a higher computational cost, especially in the 2-qubit classifier. COBYLA emerges as a balanced option, offering good accuracy with minimal computational time, particularly in the entangled system. This analysis underscores the importance of choosing appropriate optimization methods and leveraging entanglement to enhance the performance of quantum classification tasks.

Supplementary Note 6: Method

Quantum computing manipulates quantum systems to enhance information processing, leveraging superposition to simultaneously operate on multiple states for faster and more complex computation. At its core is the qubit, represented in a two-dimensional Hilbert space, with operations governed by quantum gates. These gates, essential for altering quantum states, must be unitary to ensure the conservation of probability, a fundamental principle of quantum dynamics².

The framework of a quantum circuit unfolds in three key phases: encoding classical data into quantum format, manipulating the quantum state using quantum gates, and measuring the quantum state post-transformation. This process transitions from preparing an initial quantum state, through strategic alterations via quantum gates affecting computation outcomes, to a final probabilistic measurement—distinguishing quantum computing's potential and challenges from deterministic classical computing.

Achieving optimal performance in quantum computing requires a nuanced understanding of these phases, including the initial state preparation, the strategic selection and application of quantum gates, and the final measurement process. Each component must be meticulously optimized to perform specific tasks, such as classification, highlighting the intricate interplay between quantum mechanics and computational logic in the design and execution of quantum algorithms.

A. RE-UPLOADING CLASSICAL INFORMATION AND PROCESSING

The integration of classical information into quantum computing represents a groundbreaking approach to data processing and analysis. This process begins with the strategic encoding of data into the initial wave function's coefficients within a quantum circuit³. In simpler terms, data is initially uploaded through the manipulation of qubits via rotational operations on a computational basis. This foundational step sets the stage for executing sophisticated quantum algorithms, including those designed for classification tasks.

The most successful programming paradigm in machine learning is predicated on artificial neural networks, which represent a highly abstracted and simplified model inspired by the human brain⁴. An artificial neural network comprises interconnected units or nodes known as artificial neurons, often arranged in layers⁵. These networks are characterized by their diverse architectures and the ability to learn from data through the adjustment of a vast network of parameters during the training phase. Among the various types of neural networks, feed-forward neural networks exemplify the process of sequential data processing, where input data is transformed layer by layer, simulating a form of data re-uploading at each neuron. This mechanism of data re-uploading and processing in ANNs provides a parallel to the innovative approach of constructing a universal quantum classifier using a single qubit. The essence of this quantum computing strategy lies in the repeated introduction of classical data at different stages of computation, analogous to the data processing in a single hidden layer neural network. This process can be visualized diagrammatically, as shown in figure 14 in the main paper. The neural network architecture is depicted, where data points are fed into individual processing units, analogous to neurons within the hidden layer. These neurons collectively process these input data, culminating in the activation of a final neuron responsible for constructing the output for subsequent analysis. Similarly, in the quantum domain, the single-qubit classifier incorporates data points into each stage of the computation through unitary rotations. These rotations are not isolated; rather, each one builds upon the transformations applied by its predecessors, effectively integrating the input data multiple times throughout the computation. The culmination of this process is a quantum state that encapsulates the computational outcome.

To construct a universal quantum classifier with only a single qubit, a complex integration of data input and computational processing within a single quantum circuit is crucial. We achieve this objective through the deployment of parametrized quantum circuits (PQCs). In these circuits, certain rotational angles are meticulously adjusted based on

classical parameters, which are refined through an optimization process aimed at minimizing a specifically defined cost function.

The cost function plays a pivotal role in the operational efficacy of the quantum classifier. It quantitatively assesses the circuit's performance in segregating data points into distinct categories, which are represented as separate regions on the Bloch sphere. Each of these regions corresponds to a different class, and the classifier's goal is to assign data points to the correct class based on their features.

B. Dataset Generation Methodology

In this section, we provide a detailed and standardized description of how both fixed and random datasets were constructed and evaluated throughout the study.

- **Sampling Distribution and Dimensionality:**

All data points were sampled independently and uniformly from the interval $[-1,1]^2$, corresponding to the two-dimensional input space used in all classification problems. The sampling was performed using `np.random.rand(2)` and scaled via the transformation $x \mapsto 2x - 1$ to ensure full coverage of the $[-1,1]$ range along both axes.

- **Class Balance and Geometric Design:**

We carefully selected geometric parameters to maintain balanced class distributions. In the circle classification task (non-LCP), we used a radius of $r = \sqrt{2/\pi}$ such that the area inside and outside the circle is equal, yielding a 50/50 class distribution. For the linear classification task (LCP), we defined the decision boundary as $x_1=x_2$, which symmetrically divides the domain $[-1,1]^2$ and likewise ensures class balance by design.

- **Reproducibility and Standardization:**

To ensure consistency across experiments, we fixed the random seed at 30 for all fixed dataset runs. The training set sizes varied from 1 to 200 samples depending on model complexity, while each test set consisted of 4000 uniformly sampled points. For randomized datasets, we deliberately omitted the use of a fixed seed, ensuring that each of the 20 iterations generated a new sample set from the same distribution. This approach allowed us to test the classifier's generalization ability and robustness under different data realizations. Accuracy and runtime were averaged across these 20 independent runs to obtain statistically meaningful results.

- **Dataset Types and Parameters:**

We focused on two primary classification tasks: (1) a line, representing linear separability (LCP), and (2) a circle, representing a basic non-linear separability case (non-LCP). These were chosen as fundamental and interpretable decision boundaries to evaluate the baseline performance of the quantum classifiers. All geometric parameters, such as the radius for non-LCP and the slope/intercept for LCP, were held fixed across all trials to ensure consistency and enable fair comparison across circuit designs and optimization methods.

C. Applying Cost Functions

In the realm of quantum computing, a quantum circuit is distinguished by its processing angles θ_i and associated weights w_i , leading to the generation of a final state $|\psi\rangle$. The measurement outcomes from this state are used to compute a classification error metric, defined as χ^2 . The goal is to minimize this error metric by adjusting the circuit's classical parameters, a process that can be effectively managed through various supervised machine learning techniques.

At the heart of using quantum measurement for classification tasks lies the approach of optimally aligning observed outputs with specific target classes. This alignment is primarily facilitated by the principle of maximizing orthogonality between the output states, ensuring clear distinction⁶. In the context of binary (dichotomous) classification, this means categorizing each observation into one of two predefined classes—referred to here as class A and class B. The decision criterion involves comparing the probabilities of observing the quantum state $P(0)$ for outcome 0 and $P(1)$ for outcome 1. If $P(0) > P(1)$, the observation is assigned to class A; otherwise, it falls under class B. To enhance this binary classification scheme, one can introduce a bias (λ), adjusting the threshold for classification such that observation is deemed part of class A if $P(0)$ is greater than λ , and class B if it falls below. The value of λ is chosen to maximize classification accuracy on a training dataset. The effectiveness of this approach is then confirmed through evaluation on a separate validation dataset.

Viewed through a geometric lens, the single-qubit classifier operates within a 2-dimensional Hilbert space—the Bloch sphere—where data encoding and classification decisions are delineated through specific rotational parameters. Any operation $L(i)$ is a rotation on the Bloch sphere surface. From this viewpoint, any point can be classified using just one unitary operation. Consequently, we can transfer any point to another point on the Bloch sphere by precisely selecting the rotation angles. However, when dealing with multiple data points, a single rotation may not suffice due to differing optimal rotation requirements. The solution lies in introducing additional layers into the quantum circuit, enabling distinct

rotation and fostering a richer feature map. Within this enhanced feature space, data points can be effectively segregated into their respective classes based on their positioning within the Bloch sphere's regions, thereby enabling a sophisticated and adaptable approach to quantum classification.

1) FIDELITY COST FUNCTION

The goal is to align the quantum states ($|\psi(\vec{\theta}, \vec{w}, \vec{x})\rangle$) as closely as possible to a designated target state on the Bloch sphere, as outlined in ¹. This alignment can be quantitatively assessed by measuring the angular distance between the labeled state and the data state, using the metric of relative fidelity ⁷. The primary objective focuses on maximizing the average fidelity between the quantum states produced by the circuit and the label states corresponding to their respective classes. To facilitate this, a cost function is introduced and mathematically formulated as Equation 1:

$$\chi_f^2(\vec{\theta}, \vec{w}) = \sum_{\mu=1}^M (1 - |\langle \tilde{\psi}_s | \psi(\vec{\theta}, \vec{w}, \vec{x}_\mu) \rangle|^2) \quad (1)$$

where $|\tilde{\psi}_s\rangle$ is the correct label state of the μ data point, which will correspond to one of the classes.

2) TRACE DISTANCE COST FUNCTION

In quantum information theory, quantifying the dissimilarity between two quantum states is a fundamental problem. Various distance measures have been proposed, each with its unique properties and applications. One such measure is the trace distance, which captures the distinguishability between two quantum states ⁷. Perez-Salinas et al. have analyzed the fidelity cost function with data re-uploading ¹. However, the authors do not consider the case of the trace distance cost function, which is what we focus on in this section. We will explore the definition and properties of the trace distance, particularly in the context of single-qubit systems, and discuss its potential as a cost function for quantum classifiers. Despite the different mathematical formulations of trace distance and fidelity, these two measures share many similar properties and are widely used in the quantum computing and quantum information community. However, depending on the specific application, one measure may be more convenient or easier to work with than the other. This versatility and widespread adoption of both trace distance and fidelity in the field motivates our decision to discuss and compare these two important distance measures in the context of quantum classifiers. The trace distance between quantum states ρ and σ can be defined as,

$$D(\rho, \sigma) \equiv \frac{1}{2} \text{tr} |\rho - \sigma|^2 \quad (2)$$

The trace distance between two single-qubit states, represented by their respective Bloch vectors \vec{r} and \vec{s} , is equal to one-half of the Euclidean distance between these vectors on the Bloch sphere. ⁷

$$D(\rho, \sigma) = \frac{|\vec{r} - \vec{s}|}{2}. \quad (3)$$

This relation provides a geometric interpretation of the trace distance for single-qubit systems, linking it to the intuitive notion of distance in three-dimensional space.

D. From Universality of the Single-Qubit Classifier to the Expansion into Multi-Qubit Quantum Classification

A key challenge in Quantum Machine Learning (QML) involves creating quantum circuits that efficiently handle complex tasks like classification without excessive use of quantum resources. The Universal Approximation Theorem (UAT) ⁸ is crucial for tackling this issue, demonstrating that a single-layer neural network with an appropriate activation function can approximate any continuous function to a desired accuracy, assuming enough hidden neurons are available. This UAT finds a compelling parallel in the quantum computing domain, particularly when considering the dynamics of quantum circuits. Here, the classical activation function is analogously performed by a unitary rotation acting upon a qubit. Specifically, a single-qubit quantum classifier, enhanced by the technique of data re-uploading, emerges as a universal approximator for any conceivable classification function. This universality hinges on the frequency of data re-uploading throughout the circuit's span ¹, underscoring that even a solitary qubit is capable of encoding and processing multifaceted high-dimensional data. This is achieved through the execution of multiple rotations, each characterized by distinct angles and weights. The culmination of these processes is a final quantum state, which is then analyzed against a predefined target state correlating to each class. Optimization of the circuit's parameters is pursued through the minimization of a cost function, which is indicative of the fidelity or trace distance between the comparative states.

By establishing the UAT within the context of quantum classifiers, a robust theoretical foundation is laid, alongside practical guidelines for the design and implementation of quantum circuits adept at sophisticated and non-LCP tasks with minimal quantum resource expenditure. This breakthrough not only forges a theoretical link between quantum circuits and neural networks but also paves the way for innovative approaches in QML. Through this lens, quantum circuits are envisioned not merely as computational tools but as entities with the potential to parallel, and possibly surpass, the capabilities of their classical neural network counterparts, inspiring a new wave of methodologies in the realm of QML.

To enhance the performance of the single-qubit classifier, it is proposed to extend it to a multi-qubit system. Adding more qubits, especially with entanglement, can improve the classifier's effectiveness, similar to how adding layers enhances neural networks. Entanglement may provide a quantum advantage in classification, though the analogy between multi-qubit classifiers and neural networks with entanglement is not fully understood and requires further exploration. Perez et al. propose a measurement strategy for multi-qubit classifiers, which extends the single-qubit approach. These strategies utilize a fidelity-based cost function.

E. Variational Circuit Architecture and Parameterization

To fully specify the architecture of the quantum classifier and support reproducibility, we detail here the structure of the variational circuits used in this study. The models are built using a data re-uploading framework, in which classical input data is embedded into the quantum circuit by modifying gate parameters via a linear transformation. Each circuit is composed of multiple layers; each layer includes data-dependent single-qubit gates followed by optional entanglement gates between qubits.

The primary quantum gates used are $U(\phi)$ gates, which are universal single-qubit rotation gates parameterized by three angles $\phi = (\theta, \varphi, \lambda)$. These gates are used for both trainable processing and data encoding. When entanglement is introduced, Controlled-Z (CZ) gates are applied between qubit pairs.

The parameter set for each circuit is divided into two categories:

- θ , the base rotation angles, organized as a tensor of shape (qubits, layers, 3),
- α , the data encoding weights, shaped as (qubits, layers, data dimension).

The total number of trainable parameters scales with both the number of qubits and the number of re-uploading layers. For example, the single-qubit configuration contains $3 \times \text{layers}$ trainable parameters. The two-qubit configuration without entanglement uses two parallel $U(\phi)$ gates per layer (one on each qubit), resulting in $6 \times \text{layers}$ parameters. When entanglement is included, the same number of $U(\phi)$ gates are used, along with $(\text{layers}-1)$ Controlled-Z gates placed between adjacent qubit layers.

The data encoding follows the transformation $\theta_{\text{encoded}} = \theta + \alpha \otimes x$, where x is the input feature vector. This allows the same circuit structure to dynamically adapt to different input data points while preserving trainable components.

Class label encoding differs based on the cost function used. For fidelity-based classification, labels are represented as computational basis states such as $|0\rangle$ or $|1\rangle$. For trace-distance-based classification, target class states are defined using Bloch sphere coordinates.

Supplementary Note 7: Optimization Methods

In practice, deploying a parameterized quantum classifier involves a process of minimizing within the parameter space that delineates the circuit's configuration. The process is often termed a hybrid algorithm, denoting the symbiotic relationship and advantages derived from combining quantum logic and classical logic. In particular, the ensemble of angles (θ_i) and weights (w_i) defines a parameter space that requires systematic exploration to achieve the minimization of χ^2 .

The occurrence of local minima is unavoidable. The arrangement of rotation gates results in an intricate multiplication of independent trigonometric functions, suggesting that our problem is characterized by a widespread distribution of minima.

The primary challenge boils down to minimizing a function that is defined by a vast array of parameters. In the case of a single-qubit classifier, the total number of parameters can be expressed as, where represents the problem's dimension (that is, the dimension of), and signifies the number of layers. Among these parameters, three are rotational angles, while the rest pertain to the weight [1]. To identify the most effective solution, we evaluate the performance of four distinct minimization techniques: the L-BFGS-B method, the COBYLA method, the Nelder-Mead method, and the Sequential Least Squares Programming (SLSQP) method.

The key challenge in optimizing a single-qubit classifier involves minimizing a function across a complex parameter space, calculated as $(3 + d)N$, where "d" is the problem's dimension and "N" is the number of layers. Also, in addition,

we need to consider rotational angles and the weight (\vec{w}_i) corresponding to the dimension ¹. To discover the optimal solution, we delve into the efficiency of four diverse minimization strategies: the L-BFGS-B, COBYLA, Nelder-Mead, and Sequential Least Squares Programming (SLSQP) methods.

A. L-BFGS-B METHOD

The L-BFGS-B technique, part of the quasi-Newton optimization methods, refines the Broyden–Fletcher–Goldfarb–Shanno (BFGS) approach by efficiently using limited computer memory ¹⁰. Its design excels in handling optimization tasks involving numerous variables, offering a linear memory usage advantage, making it highly effective for large-scale problems ¹¹.

The L-BFGS-B method is widely recognized as a cornerstone technique across various advanced applications in the field of graphics ^{12,13}. It specializes in minimizing a scalar function of one or several variables by initiating with a preliminary estimate of the optimum value. Through iterative refinement, it progressively improves upon this initial estimate to approach an optimal solution. The method employs function derivatives to determine the direction of steepest descent and approximates the Hessian matrix (second-order derivatives) using limited memory. The parameter update rule is given by¹⁴:

$$\theta_{k+1} = \theta_k - \alpha_k H_k^{-1} \nabla f(\theta_k)$$

where θ_k is the current parameter vector, $\nabla f(\theta_k)$ is the gradient, H_k^{-1} is an approximation of the inverse Hessian, and α_k is a step size typically determined by line search. This method is particularly efficient in handling large-scale problems due to its low memory usage and fast matrix-vector multiplications.

B. CONSTRAINED OPTIMIZATION BY LINEAR APPROXIMATION METHOD

COBYLA (Constrained Optimization BY Linear Approximation) is an optimization algorithm designed to minimize a scalar objective function that depends on one or more variables, subject to constraints ^{15,16}. One of the key features of COBYLA is that it does not require the calculation of derivatives, such as gradients or Hessians, of the objective function or constraints. This makes COBYLA particularly useful in situations where the derivatives are unknown, unreliable, or computationally expensive to obtain ¹⁵. Instead of requiring gradients or Hessians, COBYLA constructs linear approximations of both the objective function and constraints within a trust region framework. At each iteration, it solves a subproblem defined by: $\min_{\theta} f(\theta)$ subject to $c_i(\theta) \geq 0$ and approximates the objective function locally as:

$$f(\theta + \Delta\theta) \approx f(\theta) + \nabla f(\theta)^T \Delta\theta$$

although $\nabla f(\theta)$ is never explicitly calculated—its effect is estimated using linear interpolation.

COBYLA has been effectively utilized in quantum computing, especially as a classical optimization routine within Variational Hybrid Quantum-Classical Algorithms (VHQCAs) ¹⁷. These algorithms employ a parameterized quantum circuit, or ansatz, which is refined through a dynamic interchange between a classical computer and a quantum device. The classical computer adjusts the ansatz's parameters to minimize a cost function, which the quantum device efficiently evaluates. Through iterative updates based on the cost function outcomes, the VHQCA aims to discover the most effective ansatz configuration for specific problems. The derivative-free characteristic of COBYLA makes it particularly advantageous for this setting, where the cost functions often lack easily computable or analytically defined derivatives.

C. NELDER-MEAD METHOD

The Nelder-Mead algorithm, introduced by John Nelder and Roger Mead in 1965, is a widely used direct search method for unconstrained optimization problems ¹⁸. The algorithm operates by maintaining a simplex of $n+1$ points in an n -dimensional space, iteratively moving the simplex toward the optimal solution through a series of transformations, including reflection, expansion, contraction, and shrinkage ¹⁸. These operations are defined as follows:

- **Reflection:**

$$\theta_r = \bar{\theta} + \alpha(\bar{\theta} - \theta_h)$$

- **Expansion:**

$$\theta_e = \bar{\theta} + \gamma(\theta_r - \bar{\theta})$$

- **Contraction:**

$$\theta_c = \bar{\theta} + \rho(\theta_r - \bar{\theta})$$

- **Shrinkage:**

$$\theta_i = \theta_l + \sigma(\theta_i - \theta_l)$$

Here, $\bar{\theta}$ is the centroid of the best n points, θ_h is the worst-performing point, and α, γ, ρ , and σ are user-defined coefficients controlling the behavior of each transformation. This method is especially effective in low-dimensional, non-

convex optimization landscapes and is widely used when the objective function is noisy, non-differentiable, or discontinuous.

Recent studies have focused on enhancing the Nelder-Mead algorithm to improve its efficiency and adaptability. Gao and Han¹⁹ proposed an implementation of the Nelder-Mead algorithm with adaptive parameters, which can automatically adjust the parameter values based on the optimization progress. This adaptive approach has been shown to improve the algorithm's convergence speed and solution quality¹⁹.

Its capacity to address problems in which derivative information is not readily accessible renders it a favorable option for numerous applications in QML. However, it is essential to conduct comprehensive evaluations to scrutinize the method's accuracy, efficiency, and sensitivity to the initial guess for each unique application^{20,21}.

D. SEQUENTIAL LEAST SQUARES PROGRAMMING METHOD

The Sequential Least Squares Programming (SLSQP) method is an optimization technique that minimizes functions while adhering to specific constraints²². It is based on Sequential Quadratic Programming (SQP), which simplifies the optimization problem into a series of smaller, more manageable quadratic subproblems. In each subproblem, a quadratic approximation of the objective function and constraints is constructed using a second-order parabolic curve to model the function's behavior near a specific point. SLSQP updates this approximation using the quasi-Newton method. Specifically, the subproblem it solves takes the form:

$$\min_{\Delta\theta} \quad \|1/2 \Delta\theta^T B_k \Delta\theta + \nabla f(\theta_k)^T \Delta\theta\|$$

subject to:

$$\begin{aligned} c_i(\theta_k) + \nabla c_i(\theta_k)^T \Delta\theta &\geq 0 \text{ (inequality constraints)} \\ h_j(\theta_k) + \nabla h_j(\theta_k)^T \Delta\theta &= 0 \text{ (equality constraints)} \end{aligned}$$

where B_k is an approximation to the Hessian of the Lagrangian, and ∇f , ∇c_i , and ∇h_j are gradients of the objective and constraint functions.

Additionally, SLSQP applies a least-squares method to solve these quadratic subproblems, striving to minimize the total squared deviations between the approximation and actual function values. This method can handle both equality and inequality constraints, including variable bounds, by integrating a penalty function that imposes additional costs for any constraint or bound violations. SLSQP ensures efficient convergence by terminating the optimization process upon meeting a predefined convergence criterion, typically related to changes in the objective function value or the gradient vector's norm. This safeguard prevents indefinite computations, ensuring timely solutions.

Local minima are common challenges in both neural networks and quantum classifiers due to their complex mathematical structures—neural networks with compounded nonlinear functions and quantum circuits with prevalent trigonometric functions. This complexity increases the likelihood of encountering local minima during optimization. Moreover, with smaller training sets, the choice of optimization method is crucial. For instance, the Nelder-Mead method is noted for its robustness, particularly its lower susceptibility to local minima.

It is also critical to recognize that minimization methods are sensitive to noise, which can significantly impact their effectiveness, especially in practical quantum computing applications¹⁷.

```

1 # coding=utf-8
2 #####
3 #Quantum classifier
4 #Sara Aminpour, Mike Banad, Sarah Sharif
5 #Portera, 25th 2024
6
7 #School of Electrical and Computer Engineering/Center for Quantum and Technology, University of Oklahoma, Norman, OK
8 #73019 USA.
9 #####
10 #IMPORTANT NOTE: The code on the left was developed by Sara Aminpour, while the code on the right serves as the reference implementation
11 #Adrián Pérez-Salinas
12 #Additionally, our code on the left developed to analyze trace distance cost function and linear classification problem
13 #as well as necessary modification to apply OBYLAB, L-BFGS-B, NELDER-MEAD, and SLSQP minimization methods.
14 #so that the usage is automated
15 #import datetime
16 #from data_gen import data_generator
17 #from problem_gen import problem_generator, representatives, representatives_tr
18 #from fidelity_minimization import fidelity_minimization
19 #from fidelity_minimization import fidelity_minimization
20
21 from weighted_fidelity_minimization import weighted_fidelity_minimization
22 from save_data import write_summary, test_error, tester
23 from circuit import accuracy, accuracy_test, tester
24 from save_data import write_epochs_file, write_epoch, close_epochs_file, create_folder, write_epochs_error_rate
25 import numpy as np
26 from circuit import code_circuits, circuit
27 from matplotlib import cm
28 from matplotlib.colors import Normalize
29
30 def minimizer(chi, problem, qubits, entanglement, layers, method, name,
31               epochs=3000, batch_size=20, eta=0.1):
32
33     """ This function creates data and minimizes whichever problem (from the selected ones)
34     INPUT:
35         -chi: cost function, to choose between 'fidelity_chi' or 'weighted_fidelity_chi'
36         -problem: name of the problem, to choose among
37             ['circle', '3 circles', 'hypersphere', 'tricrown', 'non convex', 'crown', 'sphere', 'squares', 'wavy
38             lines']
39         -qubits: number of qubits, must be an integer
40         -entanglement: whether there is entanglement or not in the Ansatz, just 'y/n'
41         -layers: number of layers, must be an integer. If layers == 1, entanglement is not taken in account
42
43         -method: minimization method, to choose among ['SGD'], another valid for function scipy.optimize.minimize]
44
45         -name: a name we want for our files to be save with
46         -seed: numpy.random.seed for replicating results
47         -epochs: number of epochs for a 'SGD' method. If there is another method, this input has got no importance
48         -batch_size: size of the batches for stochastic gradient descent, only for 'SGD' method
49         -eta: learning rate, only for 'SGD' method
50
51     OUTPUT:
52         This function has got no outputs, but several files are saved in an appropriate folder. The files are
53             -summary.txt: Saves useful information for the problem
54             -theta.txt: saves the theta parameters as a flat array
55             -alpha.txt: saves the alpha parameters as a flat array
56             -weight.txt: saves the weights as a flat array if they exist
57
58     ..."""
59
60     data, drawing = data_generator(problem)
61
62     if problem == 'sphere':
63         train_data = data[500]
64         test_data = data[1000]
65         test_data = data[200]
66
67     if chi == 'fidelity_chi':
68         Accuracy_tr=0
69         Accuracy_te=0
70         i=1
71         while i<=2:
72             qubits_lab = qubits
73             theta, alpha, reprs = problem_generator(problem,qubits, layers, chi,
74                                         qubits_lab=qubits_lab)
75             theta, alpha, f = fidelity_minimization(theta, alpha, train_data, reprs,
76                                         entanglement, method,
77                                         batch_size, eta, epochs)
78
79             acc_train = tester(theta, alpha, train_data, reprs, entanglement, chi)
80             Acc_Train+=acc_train
81             acc_test = tester(theta, alpha, test_data, reprs, entanglement, chi)
82             Acc_Test+=acc_test
83
84             text_file_m=open('acc.txt','mode='a')
85             text_file_m.write(str(theta)+','+str(alpha)+','+str(reprs)+','+str(chi)+','+str(qubits)+','+str(layers)+','+str(entanglement)+','+str(eta)+','+str(epochs))
86             text_file_m.write(str(theta)+','+str(alpha)+','+str(reprs)+','+str(chi)+','+str(qubits)+','+str(layers)+','+str(entanglement)+','+str(eta)+','+str(epochs))
87             text_file_m.write(str(theta)+','+str(alpha)+','+str(reprs)+','+str(chi)+','+str(qubits)+','+str(layers)+','+str(entanglement)+','+str(eta)+','+str(epochs))
88             text_file_m.write(str(theta)+','+str(alpha)+','+str(reprs)+','+str(chi)+','+str(qubits)+','+str(layers)+','+str(entanglement)+','+str(eta)+','+str(epochs))
89             text_file_m.write(str(theta)+','+str(alpha)+','+str(reprs)+','+str(chi)+','+str(qubits)+','+str(layers)+','+str(entanglement)+','+str(eta)+','+str(epochs))
90             text_file_m.close()
91
92             i+=1
93             print(i-1)
94             atrAccuracy_tr+=acc_train
95             atrAccuracy_te+=acc_test
96
97         text_file_nn = open('AverageAcc.txt', mode='a')
98         text_file_nn.write(problem +','+ chi +','+ str(qubits) + 'Qubits_.' + str(layers) + 'Layers_.' + entanglement
99         + ' + method + ' + str(epochs) + 'strate' + '\n')
100        text_file_nn.write(problem +','+ chi +','+ str(qubits) + 'Qubits_.' + str(layers) + 'Layers_.' + entanglement
101        + ' + method + ' + str(epochs) + 'strate' + '\n')
102        text_file_nn.write(problem +','+ chi +','+ str(qubits) + 'Qubits_.' + str(layers) + 'Layers_.' + entanglement
103        + ' + method + ' + str(epochs) + 'strate' + '\n')
104        text_file_nn.write(problem +','+ chi +','+ str(qubits) + 'Qubits_.' + str(layers) + 'Layers_.' + entanglement
105        + ' + method + ' + str(epochs) + 'strate' + '\n')
106        text_file_nn.write(problem +','+ chi +','+ str(qubits) + 'Qubits_.' + str(layers) + 'Layers_.' + entanglement
107        + ' + method + ' + str(epochs) + 'strate' + '\n')
108        text_file_nn.close()
109
110     write_summary(chi, problem, qubits, entanglement, layers, method, name,
111                 theta, alpha, weight, f, atr, eta, epochs=epochs)
112
113     if chi == 'trace_chi':
114         Accuracy_tr=0
115         Accuracy_te=0
116         i=1
117         while i<=2:
118             qubits_lab = qubits
119             theta, alpha, reprs = problem_generator(problem,qubits, layers, chi,
120                                         qubits_lab=qubits_lab)
121             theta, alpha, f = trace_minimization(theta, alpha, train_data, reprs,
122                                         entanglement, method,
123                                         batch_size, eta, epochs)
124
125             acc_train = tester(theta, alpha, train_data, reprs, entanglement, chi)
126             Acc_Train+=acc_train
127             acc_test = tester(theta, alpha, test_data, reprs, entanglement, chi)
128             Acc_Test+=acc_test
129
130             text_file_m=open('acc.txt','mode='a')
131             text_file_m.write(str(theta)+','+str(alpha)+','+str(reprs)+','+str(chi)+','+str(qubits)+','+str(layers)+','+str(entanglement)+','+str(eta)+','+str(epochs))
132             text_file_m.write(str(theta)+','+str(alpha)+','+str(reprs)+','+str(chi)+','+str(qubits)+','+str(layers)+','+str(entanglement)+','+str(eta)+','+str(epochs))
133             text_file_m.write(str(theta)+','+str(alpha)+','+str(reprs)+','+str(chi)+','+str(qubits)+','+str(layers)+','+str(entanglement)+','+str(eta)+','+str(epochs))
134             text_file_m.write(str(theta)+','+str(alpha)+','+str(reprs)+','+str(chi)+','+str(qubits)+','+str(layers)+','+str(entanglement)+','+str(eta)+','+str(epochs))
135             text_file_m.write(str(theta)+','+str(alpha)+','+str(reprs)+','+str(chi)+','+str(qubits)+','+str(layers)+','+str(entanglement)+','+str(eta)+','+str(epochs))
136             text_file_m.write(str(theta)+','+str(alpha)+','+str(reprs)+','+str(chi)+','+str(qubits)+','+str(layers)+','+str(entanglement)+','+str(eta)+','+str(epochs))
137             text_file_m.write(str(theta)+','+str(alpha)+','+str(reprs)+','+str(chi)+','+str(qubits)+','+str(layers)+','+str(entanglement)+','+str(eta)+','+str(epochs))
138             text_file_m.close()
139
140             i+=1
141             print(i-1)
142             atrAccuracy_tr+=acc_train
143             atrAccuracy_te+=acc_test
144
145         text_file_nn = open('AverageAcc.txt', mode='a')
146         text_file_nn.write(problem +','+ chi +','+ str(qubits) + 'Qubits_.' + str(layers) + 'Layers_.' + entanglement
147         + ' + method + ' + str(epochs) + 'strate' + '\n')
148         text_file_nn.write(problem +','+ chi +','+ str(qubits) + 'Qubits_.' + str(layers) + 'Layers_.' + entanglement
149         + ' + method + ' + str(epochs) + 'strate' + '\n')
150         text_file_nn.write(problem +','+ chi +','+ str(qubits) + 'Qubits_.' + str(layers) + 'Layers_.' + entanglement
151         + ' + method + ' + str(epochs) + 'strate' + '\n')
152         text_file_nn.write(problem +','+ chi +','+ str(qubits) + 'Qubits_.' + str(layers) + 'Layers_.' + entanglement
153         + ' + method + ' + str(epochs) + 'strate' + '\n')
154         text_file_nn.write(problem +','+ chi +','+ str(qubits) + 'Qubits_.' + str(layers) + 'Layers_.' + entanglement
155         + ' + method + ' + str(epochs) + 'strate' + '\n')
156         text_file_nn.close()
157
158     write_summary(chi, problem, qubits, entanglement, layers, method, name,
159                 theta, alpha, weight, f, acc_train, acc_test, epochs=epochs)
160
161     if chi == 'weighted_fidelity_chi':
162         qubits_lab = qubits
163
164         Accuracy_tr=0
165         Accuracy_te=0
166         i=1
167         while i<=2:
168             qubits_lab = 1
169             theta, alpha, weight, reprs = problem_generator(problem,qubits, layers, chi,
170                                         qubits_lab=qubits_lab)
171             theta, alpha, weight, f = weighted_fidelity_minimization(theta, alpha, weight, train_data, reprs,
172                                         entanglement, method)
173
174             acc_train = tester(theta, alpha, train_data, reprs, entanglement, chi)
175             Acc_Train+=acc_train
176             acc_test = tester(theta, alpha, test_data, reprs, entanglement, chi)
177             Acc_Test+=acc_test
178
179             text_file_m=open('acc.txt','mode='a')
180             text_file_m.write(str(theta)+','+str(alpha)+','+str(reprs)+','+str(chi)+','+str(qubits)+','+str(layers)+','+str(entanglement)+','+str(eta)+','+str(epochs))
181             text_file_m.write(str(theta)+','+str(alpha)+','+str(reprs)+','+str(chi)+','+str(qubits)+','+str(layers)+','+str(entanglement)+','+str(eta)+','+str(epochs))
182             text_file_m.write(str(theta)+','+str(alpha)+','+str(reprs)+','+str(chi)+','+str(qubits)+','+str(layers)+','+str(entanglement)+','+str(eta)+','+str(epochs))
183             text_file_m.write(str(theta)+','+str(alpha)+','+str(reprs)+','+str(chi)+','+str(qubits)+','+str(layers)+','+str(entanglement)+','+str(eta)+','+str(epochs))
184             text_file_m.write(str(theta)+','+str(alpha)+','+str(reprs)+','+str(chi)+','+str(qubits)+','+str(layers)+','+str(entanglement)+','+str(eta)+','+str(epochs))
185             text_file_m.write(str(theta)+','+str(alpha)+','+str(reprs)+','+str(chi)+','+str(qubits)+','+str(layers)+','+str(entanglement)+','+str(eta)+','+str(epochs))
186             text_file_m.close()
187
188             i+=1
189             print(i-1)
190             atrAccuracy_tr+=acc_train
191             atrAccuracy_te+=acc_test
192
193         text_file_nn = open('AverageAcc.txt', mode='a')
194         text_file_nn.write(problem +','+ chi +','+ str(qubits) + 'Qubits_.' + str(layers) + 'Layers_.' + entanglement
195         + ' + method + ' + str(epochs) + 'strate' + '\n')
196         text_file_nn.write(problem +','+ chi +','+ str(qubits) + 'Qubits_.' + str(layers) + 'Layers_.' + entanglement
197         + ' + method + ' + str(epochs) + 'strate' + '\n')
198         text_file_nn.write(problem +','+ chi +','+ str(qubits) + 'Qubits_.' + str(layers) + 'Layers_.' + entanglement
199         + ' + method + ' + str(epochs) + 'strate' + '\n')
200         text_file_nn.write(problem +','+ chi +','+ str(qubits) + 'Qubits_.' + str(layers) + 'Layers_.' + entanglement
201         + ' + method + ' + str(epochs) + 'strate' + '\n')
202         text_file_nn.write(problem +','+ chi +','+ str(qubits) + 'Qubits_.' + str(layers) + 'Layers_.' + entanglement
203         + ' + method + ' + str(epochs) + 'strate' + '\n')
204         text_file_nn.write(problem +','+ chi +','+ str(qubits) + 'Qubits_.' + str(layers) + 'Layers_.' + entanglement
205         + ' + method + ' + str(epochs) + 'strate' + '\n')
206         text_file_nn.close()
207
208     write_summary(chi, problem, qubits, entanglement, layers, method, name,
209                 theta, alpha, weight, f, acc_train, acc_test, epochs=epochs)
210
211     if chi == 'SGD':
212         def painter(chi, problem, qubits, entanglement, layers, method, name,
213                     standard_test = True, samples = 4000, bw = False, err = False):
214             import datetime
215             now = datetime.datetime.now()
216
217             INPUT: this function takes written text files and prints the results of the problem
218             -chi: cost function, to choose between 'fidelity_chi' or 'weighted_fidelity_chi'
219             -problem: name of the problem, to choose among
220                 ['circle', '3 circles', 'hypersphere', 'tricrown', 'non convex', 'crown', 'sphere', 'squares', 'wavy
221                 lines']
222             -qubits: number of qubits, must be an integer
223             -entanglement: whether there is entanglement or not in the Ansatz, just 'y/n'
224             -layers: number of layers, must be an integer. If layers == 1, entanglement is not taken in account
225             -method: minimization method, to choose among ['SGD'], another valid for function scipy.optimize.minimize]
226             -name: a name we want for our files to be save with
227             -seed: numpy.random.seed for replicating results
228             -standard_test: Whether we want to paint the test set used for checking when minimizing. If True, seed and
229             samples = 1000, if False we want to paint the test set used for replicating results
230             -samples: number of samples of the test set
231             -bw: painting in black and white
232             OUTPUT:
233                 This function has got no outputs, but a file containing the representation of the test set is created
234
235             ..."""
236
237             if chi == 'fidelity_chi':
238                 qubits_lab = qubits
239
240                 Accuracy_tr=0
241                 Accuracy_te=0
242                 while i<=2:
243                     qubits_lab = 1
244                     theta, alpha, weight, reprs = problem_generator(problem,qubits, layers, chi,
245                                         qubits_lab=qubits_lab)
246                     theta, alpha, weight, f = fidelity_minimization(theta, alpha, weight, train_data, reprs,
247                                         entanglement, method)
248
249                     acc_train = tester(theta, alpha, train_data, reprs, entanglement, chi)
250                     Acc_Train+=acc_train
251                     acc_test = tester(theta, alpha, test_data, reprs, entanglement, chi)
252                     Acc_Test+=acc_test
253
254                     text_file_m=open('acc.txt','mode='a')
255                     text_file_m.write(str(theta)+','+str(alpha)+','+str(reprs)+','+str(chi)+','+str(qubits)+','+str(layers)+','+str(entanglement)+','+str(eta)+','+str(epochs))
256                     text_file_m.write(str(theta)+','+str(alpha)+','+str(reprs)+','+str(chi)+','+str(qubits)+','+str(layers)+','+str(entanglement)+','+str(eta)+','+str(epochs))
257                     text_file_m.write(str(theta)+','+str(alpha)+','+str(reprs)+','+str(chi)+','+str(qubits)+','+str(layers)+','+str(entanglement)+','+str(eta)+','+str(epochs))
258                     text_file_m.write(str(theta)+','+str(alpha)+','+str(reprs)+','+str(chi)+','+str(qubits)+','+str(layers)+','+str(entanglement)+','+str(eta)+','+str(epochs))
259                     text_file_m.write(str(theta)+','+str(alpha)+','+str(reprs)+','+str(chi)+','+str(qubits)+','+str(layers)+','+str(entanglement)+','+str(eta)+','+str(epochs))
260                     text_file_m.write(str(theta)+','+str(alpha)+','+str(reprs)+','+str(chi)+','+str(qubits)+','+str(layers)+','+str(entanglement)+','+str(eta)+','+str(epochs))
261                     text_file_m.close()
262
263                     i+=1
264                     print(i-1)
265                     atrAccuracy_tr+=acc_train
266                     atrAccuracy_te+=acc_test
267
268                     text_file_nn = open('AverageAcc.txt', mode='a')
269                     text_file_nn.write(problem +','+ chi +','+ str(qubits) + 'Qubits_.' + str(layers) + 'Layers_.' + entanglement
270                     + ' + method + ' + str(epochs) + 'strate' + '\n')
271                     text_file_nn.write(problem +','+ chi +','+ str(qubits) + 'Qubits_.' + str(layers) + 'Layers_.' + entanglement
272                     + ' + method + ' + str(epochs) + 'strate' + '\n')
273                     text_file_nn.write(problem +','+ chi +','+ str(qubits) + 'Qubits_.' + str(layers) + 'Layers_.' + entanglement
274                     + ' + method + ' + str(epochs) + 'strate' + '\n')
275                     text_file_nn.write(problem +','+ chi +','+ str(qubits) + 'Qubits_.' + str(layers) + 'Layers_.' + entanglement
276                     + ' + method + ' + str(epochs) + 'strate' + '\n')
277                     text_file_nn.write(problem +','+ chi +','+ str(qubits) + 'Qubits_.' + str(layers) + 'Layers_.' + entanglement
278                     + ' + method + ' + str(epochs) + 'strate' + '\n')
279                     text_file_nn.write(problem +','+ chi +','+ str(qubits) + 'Qubits_.' + str(layers) + 'Layers_.' + entanglement
280                     + ' + method + ' + str(epochs) + 'strate' + '\n')
281                     text_file_nn.close()
282
283                     write_summary(chi, problem, qubits, entanglement, layers, method, name,
284                     theta, alpha, weight, f, acc_train, acc_test, seed, epochs=epochs)
285
286                     if chi == 'weighted_fidelity_chi':
287                         qubits_lab = 1
288
289                         Accuracy_tr=0
290                         Accuracy_te=0
291                         while i<=2:
292                             qubits_lab = 1
293                             theta, alpha, weight, reprs = problem_generator(problem,qubits, layers, chi,
294                                 qubits_lab=qubits_lab)
295                             theta, alpha, weight, f = weighted_fidelity_minimization(theta, alpha, weight, train_data, reprs,
296                                 entanglement, method)
297
298                             acc_train = tester(theta, alpha, train_data, reprs, entanglement, chi)
299                             Acc_Train+=acc_train
300                             acc_test = tester(theta, alpha, test_data, reprs, entanglement, chi)
301                             Acc_Test+=acc_test
302
303                             text_file_m=open('acc.txt','mode='a')
304                             text_file_m.write(str(theta)+','+str(alpha)+','+str(reprs)+','+str(chi)+','+str(qubits)+','+str(layers)+','+str(entanglement)+','+str(eta)+','+str(epochs))
305                             text_file_m.write(str(theta)+','+str(alpha)+','+str(reprs)+','+str(chi)+','+str(qubits)+','+str(layers)+','+str(entanglement)+','+str(eta)+','+str(epochs))
306                             text_file_m.write(str(theta)+','+str(alpha)+','+str(reprs)+','+str(chi)+','+str(qubits)+','+str(layers)+','+str(entanglement)+','+str(eta)+','+str(epochs))
307                             text_file_m.write(str(theta)+','+str(alpha)+','+str(reprs)+','+str(chi)+','+str(qubits)+','+str(layers)+','+str(entanglement)+','+str(eta)+','+str(epochs))
308                             text_file_m.write(str(theta)+','+str(alpha)+','+str(reprs)+','+str(chi)+','+str(qubits)+','+str(layers)+','+str(entanglement)+','+str(eta)+','+str(epochs))
309                             text_file_m.write(str(theta)+','+str(alpha)+','+str(reprs)+','+str(chi)+','+str(qubits)+','+str(layers)+','+str(entanglement)+','+str(eta)+','+str(epochs))
310                             text_file_m.close()
311
312                             i+=1
313                             print(i-1)
314                             atrAccuracy_tr+=acc_train
315                             atrAccuracy_te+=acc_test
316
317                             text_file_nn = open('AverageAcc.txt', mode='a')
318                             text_file_nn.write(problem +','+ chi +','+ str(qubits) + 'Qubits_.' + str(layers) + 'Layers_.' + entanglement
319                             + ' + method + ' + str(epochs) + 'strate' + '\n')
320                             text_file_nn.write(problem +','+ chi +','+ str(qubits) + 'Qubits_.' + str(layers) + 'Layers_.' + entanglement
321                             + ' + method + ' + str(epochs) + 'strate' + '\n')
322                             text_file_nn.write(problem +','+ chi +','+ str(qubits) + 'Qubits_.' + str(layers) + 'Layers_.' + entanglement
323                             + ' + method + ' + str(epochs) + 'strate' + '\n')
324                             text_file_nn.write(problem +','+ chi +','+ str(qubits) + 'Qubits_.' + str(layers) + 'Layers_.' + entanglement
325                             + ' + method + ' + str(epochs) + 'strate' + '\n')
326                             text_file_nn.write(problem +','+ chi +','+ str(qubits) + 'Qubits_.' + str(layers) + 'Layers_.' + entanglement
327                             + ' + method + ' + str(epochs) + 'strate' + '\n')
328                             text_file_nn.write(problem +','+ chi +','+ str(qubits) + 'Qubits_.' + str(layers) + 'Layers_.' + entanglement
329                             + ' + method + ' + str(epochs) + 'strate' + '\n')
330                             text_file_nn.close()
331
332                             write_summary(chi, problem, qubits, entanglement, layers, method, name,
333                             theta, alpha, weight, f, acc_train, acc_test, seed, epochs=epochs)
334
335             if chi == 'SGD':
336                 def painter(chi, problem, qubits, entanglement, layers, method, name,
337                             standard_test = True, samples = 4000, bw = False, err = False):
338                     import datetime
339                     now = datetime.datetime.now()
340
341                     INPUT: this function takes written text files and prints the results of the problem
342                     -chi: cost function, to choose between 'fidelity_chi' or 'weighted_fidelity_chi'
343                     -problem: name of the problem, to choose among
344                         ['circle', '3 circles', 'hypersphere', 'tricrown', 'non convex', 'crown', 'sphere', 'squares', 'wavy
345                         lines']
346                     -qubits: number of qubits, must be an integer
347                     -entanglement: whether there is entanglement or not in the Ansatz, just 'y/n'
348                     -layers: number of layers, must be an integer. If layers == 1, entanglement is not taken in account
349                     -method: minimization method, to choose among ['SGD'], another valid for function scipy.optimize.minimize]
350                     -name: a name we want for our files to be save with
351                     -seed: numpy.random.seed for replicating results
352                     -standard_test: Whether we want to paint the test set used for checking when minimizing. If True, seed and
353                     samples = 1000, if False we want to paint the test set used for replicating results
354                     -samples: number of samples of the test set
355                     -bw: painting in black and white
356                     OUTPUT:
357                         This function has got no outputs, but a file containing the representation of the test set is created
358
359                         ..."""
360
361                         if chi == 'fidelity_chi':
362                             qubits_lab = qubits
363
364                             Accuracy_tr=0
365                             Accuracy_te=0
366                             while i<=2:
367                                 qubits_lab = 1
368                                 theta, alpha, weight, reprs = problem_generator(problem,qubits, layers, chi,
369                                     qubits_lab=qubits_lab)
370                                 theta, alpha, weight, f = fidelity_minimization(theta, alpha, weight, train_data, reprs,
371                                     entanglement, method)
372
373                                 acc_train = tester(theta, alpha, train_data, reprs, entanglement, chi)
374                                 Acc_Train+=acc_train
375                                 acc_test = tester(theta, alpha, test_data, reprs, entanglement, chi)
376                                 Acc_Test+=acc_test
377
378                                 text_file_m=open('acc.txt','mode='a')
379                                 text_file_m.write(str(theta)+','+str(alpha)+','+str(reprs)+','+str(chi)+','+str(qubits)+','+str(layers)+','+str(entanglement)+','+str(eta)+','+str(epochs))
380                                 text_file_m.write(str(theta)+','+str(alpha)+','+str(reprs)+','+str(chi)+','+str(qubits)+','+str(layers)+','+str(entanglement)+','+str(eta)+','+str(epochs))
381                                 text_file_m.write(str(theta)+','+str(alpha)+','+str(reprs)+','+str(chi)+','+str(qubits)+','+str(layers)+','+str(entanglement)+','+str(eta)+','+str(epochs))
382                                 text_file_m.write(str(theta)+','+str(alpha)+','+str(reprs)+','+str(chi)+','+str(qubits)+','+str(layers)+','+str(entanglement)+','+str(eta)+','+str(epochs))
383                                 text_file_m.write(str(theta)+','+str(alpha)+','+str(reprs)+','+str(chi)+','+str(qubits)+','+str(layers)+','+str(entanglement)+','+str(eta)+','+str(epochs))
384                                 text_file_m.write(str(theta)+','+str(alpha)+','+str(reprs)+','+str(chi)+','+str(qubits)+','+str(layers)+','+str(entanglement)+','+str(eta)+','+str(epochs))
385                                 text_file_m.close()
386
387                                 i+=1
388                                 print(i-1)
389                                 atrAccuracy_tr+=acc_train
390                                 atrAccuracy_te+=acc_test
391
392
```

```

1 # coding=utf-8
2 ##### Quantum classifier #####
3 #Sara Aminpour, Mike Banad, Sarah Sharif
4 #September 25th 2024
5
6
7 #School of Electrical and Computer Engineering/ Center for Quantum and Technology, University of Oklahoma,
8 Norman, OK 73019 USA,
9 #####
10 #The code on the left was developed by Sara Aminpour, while the code on the right serves as the reference
11 #implementation by Adrián Pérez-Salinas.
12 #The code on the left has been restructured to handle random data. So some certain sections has been deleted from
13 #the reference code.
14 #Additionally, our code on the left developed to analyze trace distance cost function and linear classification
15 #problem
16 #as well as necessary modification to apply COBYLA, L-BFGS-B, NELDER-MEAD, and SLSQP minimization methods.
17 #####
18
19
20 import numpy as np
21
22 problems = ['circle', 'line', '3 circles', 'wavy circle', 'hypersphere', 'tricrown', 'non convex', 'crown',
23 'sphere', 'squares', 'wavy lines']
24
25 def data_generator(problem, samples=None):
26     """
27         This function generates the data for a problem
28         INPUT:
29             -problem: Name of the problem, one of: 'circle', '3 circles', 'hypersphere', 'tricrown', 'non convex',
30             'crown', 'sphere', 'squares', 'wavy lines'
31             -samples Number of samples for the data
32         OUTPUT:
33             -data: set of training and test data
34             -settings: things needed for drawing
35     """
36     problem = problem.lower()
37     if problem not in problems:
38         raise ValueError('problem must be one of {}'.format(problems))
39     if samples == None:
40         if problem == 'sphere':
41             samples = 4500
42         elif problem == 'hypersphere':
43             samples = 5000
44         else:
45             samples = 4250
46
47     if problem == 'circle':
48         data, settings = _circle(samples)
49
50     if problem == '3 circles':
51         data, settings = _3_circles(samples)
52
53     if problem == 'wavy lines':
54         data, settings = _wavy_lines(samples)
55
56     if problem == 'squares':
57         data, settings = _squares(samples)
58
59     if problem == 'sphere':
60         data, settings = _sphere(samples)
61
62     if problem == 'non convex':
63         data, settings = _non_convex(samples)
64
65     if problem == 'crown':
66         data, settings = _crown(samples)
67
68     if problem == 'tricrown':
69         data, settings = _tricrown(samples)
70
71     if problem == 'hypersphere':
72         data, settings = _hypersphere(samples)
73
74 #####
75
76     return data, settings
77
78 def _circle(samples):
79     centers = np.array([[0, 0]])
80     radii = np.array([np.sqrt(2/np.pi)])
81     data = []
82     dim = 2
83     for i in range(samples):
84         x = 2 * (np.random.rand(dim)) - 1
85         y = 0
86         for c, r in zip(centers, radii):
87             if np.linalg.norm(x - c) < r:
88                 y = 1
89         data.append([x, y])
90
91     return data, (centers, radii)
92
93 def _3_circles(samples):
94     centers = np.array([-1, 1], [1, 0], [-.5, -.5])
95     radii = np.array([1, np.sqrt(6/np.pi - 1), 1/2])
96     data = []
97     dim = 2
98     for i in range(samples):
99         x = 2 * (np.random.rand(dim)) - 1
100        y = 0
101        for j, (c, r) in enumerate(zip(centers, radii)):
102            if np.linalg.norm(x - c) < r:
103                y = j + 1
104        data.append([x, y])
105
106    return data, (centers, radii)
107
108 def _wavy_lines(samples, freq = 1):
109     def fun(s):
110         return s + np.sin(freq * np.pi * s)
111     data = []
112     dim=2
113     for i in range(samples):
114         x = 2 * (np.random.rand(dim)) - 1
115         if x[1] < fun(x[0]) and x[1] < fun2(x[0]): y = 0
116         if x[1] < fun(x[0]) and x[1] > fun2(x[0]): y = 1
117         if x[1] > fun(x[0]) and x[1] < fun2(x[0]): y = 2
118         if x[1] > fun(x[0]) and x[1] > fun2(x[0]): y = 3
119         data.append([x, y])
120
121    return data, freq
122
123 def _squares(samples):
124     data = []
125     dim=2
126     for i in range(samples):
127         x = 2 * (np.random.rand(dim)) - 1
128         if x[0] < 0 and x[1] < 0: y = 0
129         if x[0] < 0 and x[1] > 0: y = 1
130         if x[0] > 0 and x[1] < 0: y = 2
131         if x[0] > 0 and x[1] > 0: y = 3
132         data.append([x, y])
133
134    return data, None
135
136 #####
137 def _line(samples):
138     data=[]
139     dim=2
140     for i in range(samples):
141         x = 2 * np.random.rand(dim) -1
142         #x = np.random.rand(dim)
143         if x[0] < x[1] : y = 0
144         if x[0] > x[1] : y = 1
145
146         data.append([x, y])
147
148    return data, None
149
150 #####
151
152 def _non_convex(samples, freq = 1, x_val = 2, sin_val = 1.5):
153     def fun(s):
154         return -x_val * s + sin_val * np.sin(freq * np.pi * s)
155
156     data = []
157     dim = 2
158     for i in range(samples):
159         x = 2 * (np.random.rand(dim)) - 1
160         if x[1] < fun(x[0]): y = 0
161         if x[1] > fun(x[0]): y = 1
162         data.append([x, y])
163
164    return data, (freq, x_val, sin_val)
165
166 def _crown(samples):
167     c = [[0,0],[0,0]]
168     r = [np.sqrt(.8), np.sqrt(.8 - 2/np.pi)]
169     data = []
170     dim = 2
171     for i in range(samples):
172         x = 2 * (np.random.rand(dim)) - 1
173         if np.linalg.norm(x - c[0]) < r[0] and np.linalg.norm(x - c[1]) > r[1]:
174             y = 1
175         else:
176             y=0
177         data.append([x, y])
178
179    return data, (c, r)
180
181 def _tricrown(samples):
182     centers = [[0,0],[0,0],[0,0]]
183     radii = [np.sqrt(.8 - 2/np.pi), np.sqrt(.8 - 2/np.pi), np.sqrt(.8)]
184     data = []
185     dim = 2
186     for i in range(samples):
187         x = 2 * (np.random.rand(dim)) - 1
188         if np.linalg.norm(x - c[0]) < r[0] and np.linalg.norm(x - c[1]) > r[1] and np.linalg.norm(x - c[2]) < r[2]:
189             y = 1
190         else:
191             y=0
192         data.append([x, y])
193
194    return data, (centers, radii)
195
196 def _sphere(samples):
197     centers = np.array([0, 0, 0])
198     radii = np.array([(3/np.pi)**(1/3)])
199     data = []
200     dim = 3
201     for i in range(samples):
202         x = 2 * (np.random.rand(dim)) - 1
203         y = 0
204         for c, r in zip(centers, radii):
205             if np.linalg.norm(x - c) < r:
206                 y = 1
207         data.append([x, y])
208
209    return data, (centers, radii)
210
211 def _hypersphere(samples):
212     centers = np.array([0, 0, 0, 0])
213     radii = np.array([(2/np.pi)**(1/2)])
214     data = []
215     dim = 4
216     for i in range(samples):
217         x = 2 * (np.random.rand(dim)) - 1
218         y = 0
219         for c, r in zip(centers, radii):
220             if np.linalg.norm(x - c) < r:
221                 y = 1
222         data.append([x, y])
223
224    return data, (centers, radii)
225
226 #####
227
228 #<>>> 3
229 #<>>> 4
230 #<>>> 5
231 #<>>> 6
232 #<>>> 7
233 #<>>> 8
234 #<>>> 9
235 #<>>> 10
236 #<>>> 11
237 #<>>> 12
238 #<>>> 13
239 #<>>> 14
240 #<>>> 15
241 #<>>> 16
242 #<>>> 17
243 #<>>> 18
244 #<>>> 19
245 #<>>> 20
246 #<>>> 21
247 #<>>> 22
248 #<>>> 23
249 #<>>> 24
250 #<>>> 25
251 #<>>> 26
252 #<>>> 27
253 #<>>> 28
254 #<>>> 29
255 #<>>> 30
256 #<>>> 31
257 #<>>> 32
258 #<>>> 33
259 #<>>> 34
260 #<>>> 35
261 #<>>> 36
262 #<>>> 37
263 #<>>> 38
264 #<>>> 39
265 #<>>> 40
266 #<>>> 41
267 #<>>> 42
268 #<>>> 43
269 #<>>> 44
270 #<>>> 45
271 #<>>> 46
272 #<>>> 47
273 #<>>> 48
274 #<>>> 49
275 #<>>> 50
276 #<>>> 51
277 #<>>> 52
278 #<>>> 53
279 #<>>> 54
280 #<>>> 55
281 #<>>> 56
282 #<>>> 57
283 #<>>> 58
284 #<>>> 59
285 #<>>> 60
286 #<>>> 61
287 #<>>> 62
288 #<>>> 63
289 #<>>> 64
290 #<>>> 65
291 #<>>> 66
292 #<>>> 67
293 #<>>> 68
294 #<>>> 69
295 #<>>> 70
296 #<>>> 71
297 #<>>> 72
298 #<>>> 73
299 #<>>> 74
300 #<>>> 75
301 #<>>> 76
302 #<>>> 77
303 #<>>> 78
304 #<>>> 79
305 #<>>> 80
306 #<>>> 81
307 #<>>> 82
308 #<>>> 83
309 #<>>> 84
310 #<>>> 85
311 #<>>> 86
312 #<>>> 87
313 #<>>> 88
314 #<>>> 89
315 #<>>> 90
316 #<>>> 91
317 #<>>> 92
318 #<>>> 93
319 #<>>> 94
320 #<>>> 95
321 #<>>> 96
322 #<>>> 97
323 #<>>> 98
324 #<>>> 99
325 #<>>> 100
326 #<>>> 101
327 #<>>> 102
328 #<>>> 103
329 #<>>> 104
330 #<>>> 105
331 #<>>> 106
332 #<>>> 107
333 #<>>> 108
334 #<>>> 109
335 #<>>> 110
336 #<>>> 111
337 #<>>> 112
338 #<>>> 113
339 #<>>> 114
340 #<>>> 115
341 #<>>> 116
342 #<>>> 117
343 #<>>> 118
344 #<>>> 119
345 #<>>> 120
346 #<>>> 121
347 #<>>> 122
348 #<>>> 123
349 #<>>> 124
350 #<>>> 125
351 #<>>> 126
352 #<>>> 127
353 #<>>> 128
354 #<>>> 129
355 #<>>> 130
356 #<>>> 131
357 #<>>> 132
358 #<>>> 133
359 #<>>> 134
360 #<>>> 135
361 #<>>> 136
362 #<>>> 137
363 #<>>> 138
364 #<>>> 139
365 #<>>> 140
366 #<>>> 141
367 #<>>> 142
368 #<>>> 143
369 #<>>> 144
370 #<>>> 145
371 #<>>> 146
372 #<>>> 147
373 #<>>> 148
374 #<>>> 149
375 #<>>> 150
376 #<>>> 151
377 #<>>> 152
378 #<>>> 153
379 #<>>> 154
380 #<>>> 155
381 #<>>> 156
382 #<>>> 157
383 #<>>> 158
384 #<>>> 159
385 #<>>> 160
386 #<>>> 161
387 #<>>> 162
388 #<>>> 163
389 #<>>> 164
390 #<>>> 165
391 #<>>> 166
392 #<>>> 167
393 #<>>> 168
394 #<>>> 169
395 #<>>> 170
396 #<>>> 171
397 #<>>> 172
398 #<>>> 173
399 #<>>> 174
400 #<>>> 175
401 #<>>> 176
402 #<>>> 177
403 #<>>> 178
404 #<>>> 179
405 #<>>> 180
406 #<>>> 181
407 #<>>> 182
408 #<>>> 183
409 #<>>> 184
410 #<>>> 185
411 #<>>> 186
412 #<>>> 187
413 #<>>> 188
414 #<>>> 189
415 #<>>> 190
416 #<>>> 191
417 #<>>> 192
418 #<>>> 193
419 #<>>> 194
420 #<>>> 195
421 #<>>> 196
422 #<>>> 197
423 #<>>> 198
424 #<>>> 199
425 #<>>> 200
426 #<>>> 201
427 #<>>> 202
428 #<>>> 203
429 #<>>> 204
430 #<>>> 205
431 #<>>> 206
432 #<>>> 207
433 #<>>> 208
434 #<>>> 209
435 #<>>> 210
436 #<>>> 211
437 #<>>> 212
438 #<>>> 213
439 #<>>> 214
440 #<>>> 215
441 #<>>> 216
442 #<>>> 217
443 #<>>> 218
444 #<>>> 219
445 #<>>> 220
446 #<>>> 221
447 #<>>> 222
448 #<>>> 223
449 #<>>> 224
450 #<>>> 225
451 #<>>> 226
452 #<>>> 227
453 #<>>> 228
454 #<>>> 229
455 #<>>> 230
456 #<>>> 231
457 #<>>> 232
458 #<>>> 233
459 #<>>> 234
460 #<>>> 235
461 #<>>> 236
462 #<>>> 237
463 #<>>> 238
464 #<>>> 239
465 #<>>> 240
466 #<>>> 241
467 #<>>> 242
468 #<>>> 243
469 #<>>> 244
470 #<>>> 245
471 #<>>> 246
472 #<>>> 247
473 #<>>> 248
474 #<>>> 249
475 #<>>> 250
476 #<>>> 251
477 #<>>> 252
478 #<>>> 253
479 #<>>> 254
480 #<>>> 255
481 #<>>> 256
482 #<>>> 257
483 #<>>> 258
484 #<>>> 259
485 #<>>> 260
486 #<>>> 261
487 #<>>> 262
488 #<>>> 263
489 #<>>> 264
490 #<>>> 265
491 #<>>> 266
492 #<>>> 267
493 #<>>> 268
494 #<>>> 269
495 #<>>> 270
496 #<>>> 271
497 #<>>> 272
498 #<>>> 273
499 #<>>> 274
500 #<>>> 275
501 #<>>> 276
502 #<>>> 277
503 #<>>> 278
504 #<>>> 279
505 #<>>> 280
506 #<>>> 281
507 #<>>> 282
508 #<>>> 283
509 #<>>> 284
510 #<>>> 285
511 #<>>> 286
512 #<>>&
```



```

1 # coding=utf-8
2 ##########
3 #Quantum classifier
4 #Sara Aminpour, Mike Banad, Sarah Sharif
5 #September 25th 2024
6
7 #School of Electrical and Computer Engineering/ Center for Quantum and Technology, University of Oklahoma, Norman, OK 73019 USA.
8 ##########
9 #IMPORTANT NOTE:
10 #The code on the left was developed by Sara Aminpour, while the code on the right serves as the reference implementation by Adrián Pérez-Salinas.
11 #The code on the left has been restructured to handle random data. So some certain sections has been deleted from the reference code.
12 #Additionally, our code on the left developed to analyze trace distance cost function and linear classification problem as well as necessary modification to apply COBYLA, L-BFGS-B, NELDER-MEAD, and SLSQP minimization methods.
13 ##########
14 #####
15 ## This file creates the problems and their settings
16 import numpy as np
17
18 def problem_generator(problem, qubits, layers, chi, qubits_lab=1):
19     """
20     This function generates everything needed for solving the problem
21     INPUT:
22         -chi: cost function, to choose between 'fidelity_chi' or 'weighted_fidelity_chi'
23         -problem: name of the problem, to choose among
24             ['circle', '3 circles', 'hypersphere', 'tricrown', 'non convex', 'crown', 'sphere', 'squares', 'wavy
25             lines']
26         -qubits: number of qubits, must be an integer
27         -layers: number of layers, must be an integer. If layers == 1, entanglement is not taken in account
28
29     OUTPUT:
30         -theta: set of parameters needed for the circuit. It is an array with shape (qubits, layers, 3)
31         -alpha: set of parameters needed for the circuit. It is an array with shape (qubits, layers, dimension of
32         data)
33         -weight: set of parameters needed for the circuit only if chi == 'weighted_fidelity_chi'. It is an array
34         with shape (classes, qubits)
35         -reprs: variable encoding the label states of the different classes
36
37     chi = chi.lower()
38     if chi in ['fidelity', 'weighted_fidelity', 'trace']: chi += '_chi'
39     if chi not in ['fidelity_chi', 'weighted_fidelity_chi', 'trace_chi']:
40         raise ValueError('Figure of merit is not valid')
41
42     if chi == 'weighted_fidelity_chi' and qubits_lab != 1:
43         qubits_lab = 1
44         print('WARNING: number of qubits for the label states has been changed to 1')
45
46     problem = problem.lower()
47     if problem == 'circle':
48         theta, alpha, reprs = _circle(qubits, layers, qubits_lab, chi)
49     elif problem == '3 circles':
50         theta, alpha, reprs = _3_circles(qubits, layers, qubits_lab, chi)
51     elif problem == 'wavy lines':
52         theta, alpha, reprs = _wavy_lines(qubits, layers, qubits_lab, chi)
53     elif problem == 'squares':
54         theta, alpha, reprs = _squares(qubits, layers, qubits_lab, chi)
55     elif problem == 'sphere':
56         theta, alpha, reprs = _sphere(qubits, layers, qubits_lab, chi)
57     elif problem == 'non convex':
58         theta, alpha, reprs = _non_convex(qubits, layers, qubits_lab, chi)
59     elif problem == 'crown':
60         theta, alpha, reprs = _crown(qubits, layers, qubits_lab, chi)
61     elif problem == 'tricrown':
62         theta, alpha, reprs = _tricrown(qubits, layers, qubits_lab, chi)
63     elif problem == 'hypersphere':
64         theta, alpha, reprs = _hypersphere(qubits, layers, qubits_lab, chi)
65
66     elif problem == 'line':
67         theta, alpha, reprs = _line(qubits, layers, qubits_lab, chi)
68 ##########
69
70     else:
71         raise ValueError('Problem is not valid')
72
73     if chi == 'fidelity_chi':
74         return theta, alpha, reprs
75
76     elif chi == 'trace_chi':
77         return theta, alpha, reprs
78
79     elif chi == 'weighted_fidelity_chi':
80         weights = np.ones((len(reprs), qubits))
81         return theta, alpha, weights, reprs
82
83     #All these are auxiliary functions for problem_generator
84     def _circle(qubits, layers, qubits_lab, chi):
85         classes = 2
86
87         if chi == 'trace_chi':
88             reprs = representatives_tr(classes, qubits_lab)
89         else:
90             reprs = representatives(classes, qubits_lab)
91
92         theta = np.random.rand(qubits, layers, 3)
93         alpha = np.random.rand(qubits, layers, 2)
94         return theta, alpha, reprs
95
96     def _3_circles(qubits, layers, qubits_lab, chi):
97         classes = 4
98
99         if chi == 'trace_chi':
100            reprs = representatives_tr(classes, qubits_lab)
101        else:
102            reprs = representatives(classes, qubits_lab)
103
104         theta = np.random.rand(qubits, layers, 3)
105         alpha = np.random.rand(qubits, layers, 2)
106         return theta, alpha, reprs
107
108     def _wavy_lines(qubits, layers, qubits_lab, chi):
109         classes = 4
110
111         if chi == 'trace_chi':
112            reprs = representatives_tr(classes, qubits_lab)
113        else:
114            reprs = representatives(classes, qubits_lab)
115
116         theta = np.random.rand(qubits, layers, 3)
117         alpha = np.random.rand(qubits, layers, 2)
118         return theta, alpha, reprs
119
120     def _squares(qubits, layers, qubits_lab, chi):
121         classes = 4
122
123         if chi == 'trace_chi':
124            reprs = representatives_tr(classes, qubits_lab)
125        else:
126            reprs = representatives(classes, qubits_lab)
127
128         theta = np.random.rand(qubits, layers, 3)
129         alpha = np.random.rand(qubits, layers, 2)
130         return theta, alpha, reprs
131
132     def _non_convex(qubits, layers, qubits_lab, chi):
133         classes = 2
134
135         if chi == 'trace_chi':
136            reprs = representatives_tr(classes, qubits_lab)
137        else:
138            reprs = representatives(classes, qubits_lab)
139
140         theta = np.random.rand(qubits, layers, 3)
141         alpha = np.random.rand(qubits, layers, 2)
142         return theta, alpha, reprs
143
144     def _crown(qubits, layers, qubits_lab, chi):
145         classes = 3
146
147         if chi == 'trace_chi':
148            reprs = representatives_tr(classes, qubits_lab)
149        else:
150            reprs = representatives(classes, qubits_lab)
151
152         theta = np.random.rand(qubits, layers, 3)
153         alpha = np.random.rand(qubits, layers, 2)
154         return theta, alpha, reprs
155
156     def _tricrown(qubits, layers, qubits_lab, chi):
157         classes = 3
158
159         if chi == 'trace_chi':
160            reprs = representatives_tr(classes, qubits_lab)
161        else:
162            reprs = representatives(classes, qubits_lab)
163
164         theta = np.random.rand(qubits, layers, 3)
165         alpha = np.random.rand(qubits, layers, 2)
166         return theta, alpha, reprs
167
168     def _sphere(qubits, layers, qubits_lab, chi):
169         classes = 2
170
171         if chi == 'trace_chi':
172            reprs = representatives_tr(classes, qubits_lab)
173        else:
174            reprs = representatives(classes, qubits_lab)
175
176         theta = np.random.rand(qubits, layers, 3)
177         alpha = np.random.rand(qubits, layers, 2)
178         return theta, alpha, reprs
179
180     def _hypersphere(qubits, layers, qubits_lab, chi):
181         classes = 2
182
183         if chi == 'trace_chi':
184            reprs = representatives_tr(classes, qubits_lab)
185        else:
186            reprs = representatives(classes, qubits_lab)
187
188         theta = np.random.rand(qubits, layers, 3)
189         alpha = np.random.rand(qubits, layers, 2)
190         return theta, alpha, reprs
191
192     def representatives_tr(classes, qubits_lab):
193         """
194         This function creates the label states for the classification task
195         INPUT:
196             -classes: number of classes of our problem
197             -qubits_lab: how many qubits will store the labels
198         OUTPUT:
199             -reprs: the label states
200
201         #reprs = np.zeros((classes, 2**qubits_lab), dtype = 'complex')
202         #reprs = np.zeros((classes, 3), dtype = 'complex')
203         if qubits_lab == 1:
204             if classes == 0:
205                 raise ValueError('Nonsense classifier')
206             if classes == 1:
207                 if classes == 1:
208                     reprs[0] = np.array([1, 0])
209                     reprs[1] = np.array([-0.2938926261462367, -0.5090369604551273, 0.8090169943749473])
210                 if classes == 2:
211                     reprs[0] = np.array([1, 0, 0])
212                     reprs[1] = np.array([-0.2938926261462367, 0.5090369604551273, -0.8090169943749473])
213                 if classes == 3:
214                     reprs[0] = np.array([1, 0, 0, 0])
215                     reprs[1] = np.array([-0.2938926261462367, 0.5090369604551273, 0.8090169943749473])
216                     reprs[2] = np.array([1, 0, 0, 0])
217                     reprs[3] = np.array([-0.2938926261462367, -0.5090369604551273, -0.8090169943749473])
218                 if classes == 4:
219                     reprs[0] = np.array([1, 0, 0, 0])
220                     reprs[1] = np.array([-0.2938926261462367, 0.5090369604551273, 0.8090169943749473])
221                     reprs[2] = np.array([1, 0, 0, 0])
222                     reprs[3] = np.array([-0.2938926261462367, -0.5090369604551273, -0.8090169943749473])
223
224         if qubits_lab == 2:
225             if classes == 0:
226                 raise ValueError('Nonsense classifier')
227             if classes == 1:
228                 if classes == 1:
229                     reprs[0] = np.array([0.29, -0.5, 0.8])
230                     reprs[1] = np.array([-0.29, 0.5, -0.8])
231                 if classes == 2:
232                     reprs[0] = np.array([1, 0, 0, 0])
233                     reprs[1] = np.array([0, 1, 0, 0])
234                 if classes == 3:
235                     reprs[0] = np.array([1, 0, 0, 0])
236                     reprs[1] = np.array([0, 1, 0, 0])
237                     reprs[2] = np.array([0, 0, 1, 0])
238                 if classes == 4:
239                     reprs[0] = np.array([1, 0, 0, 0])
240                     reprs[1] = np.array([0, 1, 0, 0])
241                     reprs[2] = np.array([0, 0, 1, 0])
242                     reprs[3] = np.array([0, 0, 0, 1])
243
244         if qubits_lab == 3:
245             if classes == 0:
246                 raise ValueError('Nonsense classifier')
247             if classes == 1:
248                 if classes == 1:
249                     reprs[0] = np.array([0.29, -0.5, 0.8])
250                     reprs[1] = np.array([-0.29, 0.5, -0.8])
251                 if classes == 2:
252                     reprs[0] = np.array([1, 0, 0, 0])
253                     reprs[1] = np.array([0, 1, 0, 0])
254                     reprs[2] = np.array([0, 0, 1, 0])
255                     reprs[3] = np.array([0, 0, 0, 1])
256                 if classes == 3:
257                     reprs[0] = np.array([1, 0, 0, 0])
258                     reprs[1] = np.array([0, 1, 0, 0])
259                     reprs[2] = np.array([0, 0, 1, 0])
260                     reprs[3] = np.array([0, 0, 0, 1])
261
262         if qubits_lab == 4:
263             if classes == 0:
264                 raise ValueError('Nonsense classifier')
265             if classes == 1:
266                 if classes == 1:
267                     reprs[0] = np.array([0.29, -0.5, 0.8])
268                     reprs[1] = np.array([-0.29, 0.5, -0.8])
269                 if classes == 2:
270                     reprs[0] = np.array([1, 0, 0, 0])
271                     reprs[1] = np.array([0, 1, 0, 0])
272                     reprs[2] = np.array([0, 0, 1, 0])
273                     reprs[3] = np.array([0, 0, 0, 1])
274
275         if qubits_lab == 5:
276             if classes == 0:
277                 raise ValueError('Nonsense classifier')
278             if classes == 1:
279                 if classes == 1:
280                     reprs[0] = np.array([0.29, -0.5, 0.8])
281                     reprs[1] = np.array([-0.29, 0.5, -0.8])
282                 if classes == 2:
283                     reprs[0] = np.array([1, 0, 0, 0])
284                     reprs[1] = np.array([0, 1, 0, 0])
285                     reprs[2] = np.array([0, 0, 1, 0])
286                     reprs[3] = np.array([0, 0, 0, 1])
287                     reprs[4] = np.array([0, 0, 0, 1])
288
289         if qubits_lab == 6:
290             if classes == 0:
291                 raise ValueError('Nonsense classifier')
292             if classes == 1:
293                 if classes == 1:
294                     reprs[0] = np.array([0.29, -0.5, 0.8])
295                     reprs[1] = np.array([-0.29, 0.5, -0.8])
296                 if classes == 2:
297                     reprs[0] = np.array([1, 0, 0, 0])
298                     reprs[1] = np.array([0, 1, 0, 0])
299                     reprs[2] = np.array([0, 0, 1, 0])
300                     reprs[3] = np.array([0, 0, 0, 1])
301                     reprs[4] = np.array([0, 0, 0, 1])
302                     reprs[5] = np.array([0, 0, 0, 1])
303
304         if qubits_lab == 7:
305             if classes == 0:
306                 raise ValueError('Nonsense classifier')
307             if classes == 1:
308                 if classes == 1:
309                     reprs[0] = np.array([0.29, -0.5, 0.8])
310                     reprs[1] = np.array([-0.29, 0.5, -0.8])
311                 if classes == 2:
312                     reprs[0] = np.array([1, 0, 0, 0])
313                     reprs[1] = np.array([0, 1, 0, 0])
314                     reprs[2] = np.array([0, 0, 1, 0])
315                     reprs[3] = np.array([0, 0, 0, 1])
316                     reprs[4] = np.array([0, 0, 0, 1])
317                     reprs[5] = np.array([0, 0, 0, 1])
318                     reprs[6] = np.array([0, 0, 0, 1])
319
320         if qubits_lab == 8:
321             if classes == 0:
322                 raise ValueError('Nonsense classifier')
323             if classes == 1:
324                 if classes == 1:
325                     reprs[0] = np.array([0.29, -0.5, 0.8])
326                     reprs[1] = np.array([-0.29, 0.5, -0.8])
327                 if classes == 2:
328                     reprs[0] = np.array([1, 0, 0, 0])
329                     reprs[1] = np.array([0, 1, 0, 0])
330                     reprs[2] = np.array([0, 0, 1, 0])
331                     reprs[3] = np.array([0, 0, 0, 1])
332                     reprs[4] = np.array([0, 0, 0, 1])
333                     reprs[5] = np.array([0, 0, 0, 1])
334                     reprs[6] = np.array([0, 0, 0, 1])
335                     reprs[7] = np.array([0, 0, 0, 1])
336
337         if qubits_lab == 9:
338             if classes == 0:
339                 raise ValueError('Nonsense classifier')
340             if classes == 1:
341                 if classes == 1:
342                     reprs[0] = np.array([0.29, -0.5, 0.8])
343                     reprs[1] = np.array([-0.29, 0.5, -0.8])
344                 if classes == 2:
345                     reprs[0] = np.array([1, 0, 0, 0])
346                     reprs[1] = np.array([0, 1, 0, 0])
347                     reprs[2] = np.array([0, 0, 1, 0])
348                     reprs[3] = np.array([0, 0, 0, 1])
349                     reprs[4] = np.array([0, 0, 0, 1])
350                     reprs[5] = np.array([0, 0, 0, 1])
351                     reprs[6] = np.array([0, 0, 0, 1])
352                     reprs[7] = np.array([0, 0, 0, 1])
353                     reprs[8] = np.array([0, 0, 0, 1])
354
355         if qubits_lab == 10:
356             if classes == 0:
357                 raise ValueError('Nonsense classifier')
358             if classes == 1:
359                 if classes == 1:
360                     reprs[0] = np.array([0.29, -0.5, 0.8])
361                     reprs[1] = np.array([-0.29, 0.5, -0.8])
362                 if classes == 2:
363                     reprs[0] = np.array([1, 0, 0, 0])
364                     reprs[1] = np.array([0, 1, 0, 0])
365                     reprs[2] = np.array([0, 0, 1, 0])
366                     reprs[3] = np.array([0, 0, 0, 1])
367                     reprs[4] = np.array([0, 0, 0, 1])
368                     reprs[5] = np.array([0, 0, 0, 1])
369                     reprs[6] = np.array([0, 0, 0, 1])
370                     reprs[7] = np.array([0, 0, 0, 1])
371                     reprs[8] = np.array([0, 0, 0, 1])
372                     reprs[9] = np.array([0, 0, 0, 1])
373
374         if qubits_lab == 11:
375             if classes == 0:
376                 raise ValueError('Nonsense classifier')
377             if classes == 1:
378                 if classes == 1:
379                     reprs[0] = np.array([0.29, -0.5, 0.8])
380                     reprs[1] = np.array([-0.29, 0.5, -0.8])
381                 if classes == 2:
382                     reprs[0] = np.array([1, 0, 0, 0])
383                     reprs[1] = np.array([0, 1, 0, 0])
384                     reprs[2] = np.array([0, 0, 1, 0])
385                     reprs[3] = np.array([0, 0, 0, 1])
386                     reprs[4] = np.array([0, 0, 0, 1])
387                     reprs[5] = np.array([0, 0, 0, 1])
388                     reprs[6] = np.array([0, 0, 0, 1])
389                     reprs[7] = np.array([0, 0, 0, 1])
390                     reprs[8] = np.array([0, 0, 0, 1])
391                     reprs[9] = np.array([0, 0, 0, 1])
392                     reprs[10] = np.array([0, 0, 0, 1])
393
394         if qubits_lab == 12:
395             if classes == 0:
396                 raise ValueError('Nonsense classifier')
397             if classes == 1:
398                 if classes == 1:
399                     reprs[0] = np.array([0.29, -0.5, 0.8])
400                     reprs[1] = np.array([-0.29, 0.5, -0.8])
401                 if classes == 2:
402                     reprs[0] = np.array([1, 0, 0, 0])
403                     reprs[1] = np.array([0, 1, 0, 0])
404                     reprs[2] = np.array([0, 0, 1, 0])
405                     reprs[3] = np.array([0, 0, 0, 1])
406                     reprs[4] = np.array([0, 0, 0, 1])
407                     reprs[5] = np.array([0, 0, 0, 1])
408                     reprs[6] = np.array([0, 0, 0, 1])
409                     reprs[7] = np.array([0, 0, 0, 1])
410                     reprs[8] = np.array([0, 0, 0, 1])
411                     reprs[9] = np.array([0, 0, 0, 1])
412                     reprs[10] = np.array([0, 0, 0, 1])
413
414         if qubits_lab == 13:
415             if classes == 0:
416                 raise ValueError('Nonsense classifier')
417             if classes == 1:
418                 if classes == 1:
419                     reprs[0] = np.array([0.29, -0.5, 0.8])
420                     reprs[1] = np.array([-0.29, 0.5, -0.8])
421                 if classes == 2:
422                     reprs[0] = np.array([1, 0, 0, 0])
423                     reprs[1] = np.array([0, 1, 0, 0])
424                     reprs[2] = np.array([0, 0, 1, 0])
425                     reprs[3] = np.array([0, 0, 0, 1])
426                     reprs[4] = np.array([0, 0, 0, 1])
427                     reprs[5] = np.array([0, 0, 0, 1])
428                     reprs[6] = np.array([0, 0, 0, 1])
429                     reprs[7] = np.array([0, 0, 0, 1])
430                     reprs[8] = np.array([0, 0, 0, 1])
431                     reprs[9] = np.array([0, 0, 0, 1])
432                     reprs[10] = np.array([0, 0, 0, 1])
433
434         if qubits_lab == 14:
435             if classes == 0:
436                 raise ValueError('Nonsense classifier')
437             if classes == 1:
438                 if classes == 1:
439                     reprs[0] = np.array([0.29, -0.5, 0.8])
440                     reprs[1] = np.array([-0.29, 0.5, -0.8])
441                 if classes == 2:
442                     reprs[0] = np.array([1, 0, 0, 0])
443                     reprs[1] = np.array([0, 1, 0, 0])
444                     reprs[2] = np.array([0, 0, 1, 0])
445                     reprs[3] = np.array([0, 0, 0, 1])
446                     reprs[4] = np.array([0, 0, 0, 1])
447                     reprs[5] = np.array([0, 0, 0, 1])
448                     reprs[6] = np.array([0, 0, 0, 1])
449                     reprs[7] = np.array([0, 0, 0, 1])
450                     reprs[8] = np.array([0, 0, 0, 1])
451                     reprs[9] = np.array([0, 0, 0, 1])
452                     reprs[10] = np.array([0, 0, 0, 1])
453
454         if qubits_lab == 15:
455             if classes == 0:
456                 raise ValueError('Nonsense classifier')
457             if classes == 1:
458                 if classes == 1:
459                     reprs[0] = np.array([0.29, -0.5, 0.8])
460                     reprs[1] = np.array([-0.29, 0.5, -0.8])
461                 if classes == 2:
462                     reprs[0] = np.array([1, 0, 0, 0])
463                     reprs[1] = np.array([0, 1, 0, 0])
464                     reprs[2] = np.array([0, 0, 1, 0])
465                     reprs[3] = np.array([0, 0, 0, 1])
466                     reprs[4] = np.array([0, 0, 0, 1])
467                     reprs[5] = np.array([0, 0, 0, 1])
468                     reprs[6] = np.array([0, 0, 0, 1])
469                     reprs[7] = np.array([0, 0, 0, 1])
470                     reprs[8] =
```

```

1 # coding=utf-8
2 #####Quantum classifier#####
3 #Quantum classifier
4 #Sara Aminpour, Mike Banad, Sarah Sharif
5 #September 25th 2024
6
7 #School of Electrical and Computer Engineering/ Center for Quantum and Technology, University of Oklahoma, Norman, OK 73019 USA,
8 #####Quantum classifier#####
9 #IMPORTANT_NOTE:
10 #The code on the left was developed by Sara Aminpour, while the code on the right serves as the reference implementation by Adrián
11 Pérez-Salinas.
12 #The code on the left has been restructured to handle random data. So some certain sections has been deleted from the reference
13 #code.
14 #Additionally, our code on the left developed to analyze trace distance cost function and linear classification problem
15 #as well as necessary modification to apply COBYLA, L-BFGS-B, NELDER-MEAD, and SLSQP minimization methods.
16 #####Quantum classifier#####
17 ## This is an auxiliary file. It provides the tools needed for simulating quantum
18 # circuits.
19
20 import numpy as np
21 class QCircuit(object):
22     def __init__(self,qubits):
23         self.num_qubits = qubits
24         self.psi = [0]*2**self.num_qubits
25         self.psi[0] = 1
26         self.E_x=0
27         self.E_y=0
28         self.E_z=0
29         self.r=np.array([0,0,0])
30
31     def Ry(self,i,theta):
32         if i>=self.num_qubits: raise ValueError('There are not enough qubits')
33         c = np.cos(theta/2)
34         s = np.sin(theta/2)
35         for k in range(2**self.num_qubits-1):
36             S = k%(2**i) + 2*(k - k%(2**i))
37             S_=S + 2**i
38             a=c*self.psi[S] - s*self.psi[S_];
39             b=s*self.psi[S] + c*self.psi[S_];
40             self.psi[S]=a; self.psi[S_]=b;
41
42     def Rx(self,i,theta):
43         if i>=self.num_qubits: raise ValueError('There are not enough qubits')
44         c = np.cos(theta/2)
45         s = np.sin(theta/2)
46         for k in range(2**self.num_qubits-1):
47             S = k%(2**i) + 2*(k - k%(2**i))
48             S_=S + 2**i
49             a=c*self.psi[S] - lj*s*self.psi[S_];
50             b=lj*s*self.psi[S] + c*self.psi[S_];
51             self.psi[S]=a; self.psi[S_]=b;
52
53     def U2(self,i,phi,lamb):
54         if i>=self.num_qubits: raise ValueError('There are not enough qubits')
55         f = np.exp(lj*phi)
56         l = np.exp(-lj*lamb)
57         for k in range(2**self.num_qubits-1):
58             S = k%(2**i) + 2*(k - k%(2**i))
59             S_=S + 2**i
60             a=l/np.sqrt(2)*(self.psi[S] - l*self.psi[S_]);
61             b=l/np.sqrt(2)*(f*self.psi[S] + f*l*self.psi[S_]);
62             self.psi[S]=a; self.psi[S_]=b;
63
64     def U3(self, i, theta3):
65         if i>=self.num_qubits: raise ValueError('There are not enough qubits')
66         c = np.cos(theta3[0] / 2)
67         s = np.sin(theta3[0] / 2)
68         e_phi = np.exp(lj * theta3[1] / 2)
69         e_phi_s = np.conj(e_phi)
70         e_lambda = np.exp(lj * theta3[2] / 2)
71         e_lambda_s = np.conj(e_lambda)
72
73         for k in range(2 ** (self.num_qubits - 1)):
74             S = k % (2 ** i) + 2 * (k - k % (2 ** i))
75             S_=S + 2**i
76             a = c * e_phi * e_lambda * self.psi[S] - s * e_phi * e_lambda_s * self.psi[S_];
77
78             b = s * e_phi_s * e_lambda * self.psi[S] + c * e_phi_s * e_lambda_s * self.psi[S_];
79
80             self.psi[S] = a;
81             self.psi[S_] = b;
82
83         theta_f=np.arccos(np.abs(self.psi[S])**2 - np.abs(self.psi[S_])**2) - np.pi/2
84         phi_f=np.angle(self.psi[S_] / self.psi[S])
85         self.r=np.array([np.sin(theta_f)*np.cos(phi_f),np.sin(phi_f)*np.sin(theta_f),np.cos(theta_f)])
86
87     def Rz(self,i,theta):
88         if i>=self.num_qubits: raise ValueError('There are not enough qubits')
89         ex = np.exp(lj*theta)
90         for k in range(2**self.num_qubits-1):
91             S = k%(2**i) + 2*(k - k%(2**i)) + 2**i
92             self.psi[S]=ex*self.psi[S];
93
94     def Hx(self,i):
95         if i>=self.num_qubits: raise ValueError('There are not enough qubits')
96         for k in range(2**self.num_qubits-1):
97             S = k%(2**i) + 2*(k - k%(2**i))
98             S_=S + 2**i
99             a=1/np.sqrt(2)*self.psi[S] + 1/np.sqrt(2)*self.psi[S_];
100            b=1/np.sqrt(2)*self.psi[S] - 1/np.sqrt(2)*self.psi[S_];
101            self.psi[S] = a
102            self.psi[S_] = b
103
104     def Hy(self,i):
105         if i>=self.num_qubits: raise ValueError('There are not enough qubits')
106         for k in range(2**self.num_qubits-1):
107             S = k%(2**i) + 2*(k - k%(2**i))
108             S_=S + 2**i
109             a=1/np.sqrt(2)*self.psi[S] - lj/np.sqrt(2)*self.psi[S_];
110             b=lj/np.sqrt(2)*self.psi[S] + 1/np.sqrt(2)*self.psi[S_];
111             self.psi[S] = a
112             self.psi[S_] = b
113
114     def HyT(self,i):
115         if i>=self.num_qubits: raise ValueError('There are not enough qubits')
116         for k in range(2**self.num_qubits-1):
117             S = k%(2**i) + 2*(k - k%(2**i))
118             S_=S + 2**i
119             a=1/np.sqrt(2)*self.psi[S] + 1j/np.sqrt(2)*self.psi[S_];
120             b=1j/np.sqrt(2)*self.psi[S] + 1/np.sqrt(2)*self.psi[S_];
121             self.psi[S] = a
122             self.psi[S_] = b
123
124     def Cz(self,i,j):
125         if i>=self.num_qubits: raise ValueError('There are not enough qubits')
126         if j>=self.num_qubits: raise ValueError('There are not enough qubits')
127         if i==j: raise ValueError('Control and target qubits are the same')
128         for k in range(2**self.num_qubits-2):
129             S = k%2**i +
130             ( k - k%2**i)*2**j + 2*
131             ( k - k%2**i)*2**j - 2*((k-k%2**i)%2**j)) + 2**i + 2**j;
132             self.psi[S]=self.psi[S]
133
134     def SWAP(self,i,j):
135         if i>=self.num_qubits: raise ValueError('There are not enough qubits')
136         if j>=self.num_qubits: raise ValueError('There are not enough qubits')
137         if i==j: raise ValueError('Control and target qubits are the same')
138         for k in range(2**self.num_qubits-2):
139             S = k%2**i +
140             ( k - k%2**i)*2**j + 2*
141             ( k - k%2**i)*2**j - 2*((k-k%2**i)%2**j)) + 2**j;
142             S_=S + 2**i - 2**j
143             a=self.psi[S]
144             self.psi[S_] = self.psi[S]
145             self.psi[S] = a
146             self.psi[S],self.psi[S_] = self.psi[S_],self.psi[S]
147
148     def Cx(self,i,j):
149         if i==j: control
150         if j==i: target
151         if i>=self.num_qubits: raise ValueError('There are not enough qubits')
152         if j>=self.num_qubits: raise ValueError('There are not enough qubits')
153         if i==j: raise ValueError('Control and target qubits are the same')
154         for k in range(2**self.num_qubits-2):
155             S = k%2**i +
156             ( k - k%2**i)*2**j + 2*
157             ( k - k%2**i)*2**j - 2*((k-k%2**i)%2**j)) + 2**i;
158             S_=S + 2**j
159             a=self.psi[S]
160             self.psi[S_] = self.psi[S]
161             self.psi[S] = a
162             self.psi[S],self.psi[S_] = self.psi[S_],self.psi[S]
163
164     def Cy(self,i,j):
165         if i>=self.num_qubits: raise ValueError('There are not enough qubits')
166         if j>=self.num_qubits: raise ValueError('There are not enough qubits')
167         if i==j: raise ValueError('Control and target qubits are the same')
168         for k in range(2**self.num_qubits-2):
169             S = k%2**i +
170             ( k - k%2**i)*2**j + 2*
171             ( k - k%2**i)*2**j - 2*((k-k%2**i)%2**j)) + 2**i;
172             S_=S + 2**j
173             self.psi[S],self.psi[S_] = lj*self.psi[S_],-lj*self.psi[S]
174
175     def MeasureZ(self):
176         self.E_z = 0;
177         for h in range(2 ** self.num_qubits):
178             s = np.binary_repr(h, width=self.num_qubits)
179             self.E_z += np.abs(self.psi[h])**2*(s.count('1')-s.count('0'))
180
181     def MeasureX(self):
182         self.E_x = 0;
183         for i in range(self.num_qubits):
184             self.Hx(i);
185         for h in range(2 ** self.num_qubits):
186             s = np.binary_repr(h, width=self.num_qubits)
187             self.E_x += np.abs(self.psi[h])**2*(s.count('1')-s.count('0'))
188         for i in range(self.num_qubits):
189             self.Hx(i);
190
191     def MeasureY(self):
192         self.E_y = 0;
193         for i in range(self.num_qubits):
194             self.Hy(i);
195         for h in range(2 ** self.num_qubits):
196             s = np.binary_repr(h, width=self.num_qubits)
197             self.E_y += np.abs(self.psi[h])**2*(s.count('1')-s.count('0'))
198         for i in range(self.num_qubits):
199             self.Hy(i);
200
201     def reduced_density_matrix(self, q):
202         rho = np.zeros((2,2), dtype='complex')
203         for i in range(2):
204             for j in range(i + 1):
205                 for k in range(2**self.num_qubits-1):
206                     S = k%(2**q) + 2*(k - k%(2**q))
207                     rho[i,j] += self.psi[S + i*2**q] * np.conj(self.psi[S + j*2**q])
208                     rho[j,i] = np.conj(rho[i,j])
209
210         return rho
211
212
213     def __init__(self, q):
214         rho = np.zeros((2,2), dtype='complex')
215         for i in range(2):
216             for j in range(i + 1):
217                 for k in range(2**self.num_qubits-1):
218                     S = k%(2**q) + 2*(k - k%(2**q))
219                     rho[i,j] += self.psi[S + i*2**q] * np.conj(self.psi[S + j*2**q])
220                     rho[j,i] = np.conj(rho[i,j])
221
222         return rho
223
224
225     def __init__(self, q):
226         rho = np.zeros((2,2), dtype='complex')
227         for i in range(2):
228             for j in range(i + 1):
229                 for k in range(2**self.num_qubits-1):
230                     S = k%(2**q) + 2*(k - k%(2**q))
231                     rho[i,j] += self.psi[S + i*2**q] * np.conj(self.psi[S + j*2**q])
232                     rho[j,i] = np.conj(rho[i,j])
233
234         return rho
235
236
237     def __init__(self, q):
238         rho = np.zeros((2,2), dtype='complex')
239         for i in range(2):
240             for j in range(i + 1):
241                 for k in range(2**self.num_qubits-1):
242                     S = k%(2**q) + 2*(k - k%(2**q))
243                     rho[i,j] += self.psi[S + i*2**q] * np.conj(self.psi[S + j*2**q])
244                     rho[j,i] = np.conj(rho[i,j])
245
246         return rho
247
248
249     def __init__(self, q):
250         rho = np.zeros((2,2), dtype='complex')
251         for i in range(2):
252             for j in range(i + 1):
253                 for k in range(2**self.num_qubits-1):
254                     S = k%(2**q) + 2*(k - k%(2**q))
255                     rho[i,j] += self.psi[S + i*2**q] * np.conj(self.psi[S + j*2**q])
256                     rho[j,i] = np.conj(rho[i,j])
257
258         return rho
259
260
261     def __init__(self, q):
262         rho = np.zeros((2,2), dtype='complex')
263         for i in range(2):
264             for j in range(i + 1):
265                 for k in range(2**self.num_qubits-1):
266                     S = k%(2**q) + 2*(k - k%(2**q))
267                     rho[i,j] += self.psi[S + i*2**q] * np.conj(self.psi[S + j*2**q])
268                     rho[j,i] = np.conj(rho[i,j])
269
270         return rho
271
272
273     def __init__(self, q):
274         rho = np.zeros((2,2), dtype='complex')
275         for i in range(2):
276             for j in range(i + 1):
277                 for k in range(2**self.num_qubits-1):
278                     S = k%(2**q) + 2*(k - k%(2**q))
279                     rho[i,j] += self.psi[S + i*2**q] * np.conj(self.psi[S + j*2**q])
280                     rho[j,i] = np.conj(rho[i,j])
281
282         return rho
283
284
285     def __init__(self, q):
286         rho = np.zeros((2,2), dtype='complex')
287         for i in range(2):
288             for j in range(i + 1):
289                 for k in range(2**self.num_qubits-1):
290                     S = k%(2**q) + 2*(k - k%(2**q))
291                     rho[i,j] += self.psi[S + i*2**q] * np.conj(self.psi[S + j*2**q])
292                     rho[j,i] = np.conj(rho[i,j])
293
294         return rho
295
296
297     def __init__(self, q):
298         rho = np.zeros((2,2), dtype='complex')
299         for i in range(2):
300             for j in range(i + 1):
301                 for k in range(2**self.num_qubits-1):
302                     S = k%(2**q) + 2*(k - k%(2**q))
303                     rho[i,j] += self.psi[S + i*2**q] * np.conj(self.psi[S + j*2**q])
304                     rho[j,i] = np.conj(rho[i,j])
305
306         return rho
307
308
309     def __init__(self, q):
310         rho = np.zeros((2,2), dtype='complex')
311         for i in range(2):
312             for j in range(i + 1):
313                 for k in range(2**self.num_qubits-1):
314                     S = k%(2**q) + 2*(k - k%(2**q))
315                     rho[i,j] += self.psi[S + i*2**q] * np.conj(self.psi[S + j*2**q])
316                     rho[j,i] = np.conj(rho[i,j])
317
318         return rho
319
320
321     def __init__(self, q):
322         rho = np.zeros((2,2), dtype='complex')
323         for i in range(2):
324             for j in range(i + 1):
325                 for k in range(2**self.num_qubits-1):
326                     S = k%(2**q) + 2*(k - k%(2**q))
327                     rho[i,j] += self.psi[S + i*2**q] * np.conj(self.psi[S + j*2**q])
328                     rho[j,i] = np.conj(rho[i,j])
329
330         return rho
331
332
333     def __init__(self, q):
334         rho = np.zeros((2,2), dtype='complex')
335         for i in range(2):
336             for j in range(i + 1):
337                 for k in range(2**self.num_qubits-1):
338                     S = k%(2**q) + 2*(k - k%(2**q))
339                     rho[i,j] += self.psi[S + i*2**q] * np.conj(self.psi[S + j*2**q])
340                     rho[j,i] = np.conj(rho[i,j])
341
342         return rho
343
344
345     def __init__(self, q):
346         rho = np.zeros((2,2), dtype='complex')
347         for i in range(2):
348             for j in range(i + 1):
349                 for k in range(2**self.num_qubits-1):
350                     S = k%(2**q) + 2*(k - k%(2**q))
351                     rho[i,j] += self.psi[S + i*2**q] * np.conj(self.psi[S + j*2**q])
352                     rho[j,i] = np.conj(rho[i,j])
353
354         return rho
355
356
357     def __init__(self, q):
358         rho = np.zeros((2,2), dtype='complex')
359         for i in range(2):
360             for j in range(i + 1):
361                 for k in range(2**self.num_qubits-1):
362                     S = k%(2**q) + 2*(k - k%(2**q))
363                     rho[i,j] += self.psi[S + i*2**q] * np.conj(self.psi[S + j*2**q])
364                     rho[j,i] = np.conj(rho[i,j])
365
366         return rho
367
368
369     def __init__(self, q):
370         rho = np.zeros((2,2), dtype='complex')
371         for i in range(2):
372             for j in range(i + 1):
373                 for k in range(2**self.num_qubits-1):
374                     S = k%(2**q) + 2*(k - k%(2**q))
375                     rho[i,j] += self.psi[S + i*2**q] * np.conj(self.psi[S + j*2**q])
376                     rho[j,i] = np.conj(rho[i,j])
377
378         return rho
379
380
381     def __init__(self, q):
382         rho = np.zeros((2,2), dtype='complex')
383         for i in range(2):
384             for j in range(i + 1):
385                 for k in range(2**self.num_qubits-1):
386                     S = k%(2**q) + 2*(k - k%(2**q))
387                     rho[i,j] += self.psi[S + i*2**q] * np.conj(self.psi[S + j*2**q])
388                     rho[j,i] = np.conj(rho[i,j])
389
390         return rho
391
392
393     def __init__(self, q):
394         rho = np.zeros((2,2), dtype='complex')
395         for i in range(2):
396             for j in range(i + 1):
397                 for k in range(2**self.num_qubits-1):
398                     S = k%(2**q) + 2*(k - k%(2**q))
399                     rho[i,j] += self.psi[S + i*2**q] * np.conj(self.psi[S + j*2**q])
400                     rho[j,i] = np.conj(rho[i,j])
401
402         return rho
403
404
405     def __init__(self, q):
406         rho = np.zeros((2,2), dtype='complex')
407         for i in range(2):
408             for j in range(i + 1):
409                 for k in range(2**self.num_qubits-1):
410                     S = k%(2**q) + 2*(k - k%(2**q))
411                     rho[i,j] += self.psi[S + i*2**q] * np.conj(self.psi[S + j*2**q])
412                     rho[j,i] = np.conj(rho[i,j])
413
414         return rho
415
416
417     def __init__(self, q):
418         rho = np.zeros((2,2), dtype='complex')
419         for i in range(2):
420             for j in range(i + 1):
421                 for k in range(2**self.num_qubits-1):
422                     S = k%(2**q) + 2*(k - k%(2**q))
423                     rho[i,j] += self.psi[S + i*2**q] * np.conj(self.psi[S + j*2**q])
424                     rho[j,i] = np.conj(rho[i,j])
425
426         return rho
427
428
429     def __init__(self, q):
430         rho = np.zeros((2,2), dtype='complex')
431         for i in range(2):
432             for j in range(i + 1):
433                 for k in range(2**self.num_qubits-1):
434                     S = k%(2**q) + 2*(k - k%(2**q))
435                     rho[i,j] += self.psi[S + i*2**q] * np.conj(self.psi[S + j*2**q])
436                     rho[j,i] = np.conj(rho[i,j])
437
438         return rho
439
440
441     def __init__(self, q):
442         rho = np.zeros((2,2), dtype='complex')
443         for i in range(2):
444             for j in range(i + 1):
445                 for k in range(2**self.num_qubits-1):
446                     S = k%(2**q) + 2*(k - k%(2**q))
447                     rho[i,j] += self.psi[S + i*2**q] * np.conj(self.psi[S + j*2**q])
448                     rho[j,i] = np.conj(rho[i,j])
449
450         return rho
451
452
453     def __init__(self, q):
454         rho = np.zeros((2,2), dtype='complex')
455         for i in range(2):
456             for j in range(i + 1):
457                 for k in range(2**self.num_qubits-1):
458                     S = k%(2**q) + 2*(k - k%(2**q))
459                     rho[i,j] += self.psi[S + i*2**q] * np.conj(self.psi[S + j*2**q])
460                     rho[j,i] = np.conj(rho[i,j])
461
462         return rho
463
464
465     def __init__(self, q):
466         rho = np.zeros((2,2), dtype='complex')
467         for i in range(2):
468             for j in range(i + 1):
469                 for k in range(2**self.num_qubits-1):
470                     S = k%(2**q) + 2*(k - k%(2**q))
471                     rho[i,j] += self.psi[S + i*2**q] * np.conj(self.psi[S + j*2**q])
472                     rho[j,i] = np.conj(rho[i,j])
473
474         return rho
475
476
477     def __init__(self, q):
478         rho = np.zeros((2,2), dtype='complex')
479         for i in range(2):
480             for j in range(i + 1):
481                 for k in range(2**self.num_qubits-1):
482                     S = k%(2**q) + 2*(k - k%(2**q))
483                     rho[i,j] += self.psi[S + i*2**q] * np.conj(self.psi[S + j*2**q])
484                     rho[j,i] = np.conj(rho[i,j])
485
486         return rho
487
488
489     def __init__(self, q):
490         rho = np.zeros((2,2), dtype='complex')
491         for i in range(2):
492             for j in range(i + 1):
493                 for k in range(2**self.num_qubits-1):
494                     S = k%(2**q) + 2*(k - k%(2**q))
495                     rho[i,j] += self.psi[S + i*2**q] * np.conj(self.psi[S + j*2**q])
496                     rho[j,i] = np.conj(rho[i,j])
497
498         return rho
499
500
501     def __init__(self, q):
502         rho = np.zeros((2,2), dtype='complex')
503         for i in range(2):
504             for j in range(i + 1):
505                 for k in range(2**self.num_qubits-1):
506                     S = k%(2**q) + 2*(k - k%(2**q))
507                     rho[i,j] += self.psi[S + i*2**q] * np.conj(self.psi[S + j*2**q])
508                     rho[j,i] = np.conj(rho[i,j])
509
510         return rho
511
512
513     def __init__(self, q):
514         rho = np.zeros((2,2), dtype='complex')
515         for i in range(2):
516             for j in range(i + 1):
517                 for k in range(2**self.num_qubits-1):
518                     S = k%(2**q) + 2*(k - k%(2**q))
519                     rho[i,j] += self.psi[S + i*2**q] * np.conj(self.psi[S + j*2**q])
520                     rho[j,i] = np.conj(rho[i,j])
521
522         return rho
523
524
525     def __init__(self, q):
526         rho = np.zeros((2,2), dtype='complex')
527         for i in range(2):
528             for j in range(i + 1):
529                 for k in range(2**self.num_qubits-1):
530                     S = k%(2**q) + 2*(k - k%(2**q))
531                     rho[i,j] += self.psi[S + i*2**q] * np.conj(self.psi[S + j*2**q])
532                     rho[j,i] = np.conj(rho[i,j])
533
534         return rho
53
```

```
Text Compare
```

```
[[{"code": "# coding=utf-8", "x": 1, "y": 1}, {"code": "#!/usr/bin/python", "x": 1, "y": 2}, {"code": "# quantum classifier", "x": 1, "y": 3}, {"code": "# Sara Amig\"or, Mike Banad, Sarah Sharif", "x": 1, "y": 4}, {"code": "# September 20th 2014", "x": 1, "y": 5}, {"code": "#", "x": 1, "y": 6}, {"code": "# School of Electrical and Computer Engineering/ Center for Quantum and Technology, University of Oklahoma, Norman, OK 73049 USA", "x": 1, "y": 7}, {"code": "# The code on the left was developed by Sara Amig\"or, while the code on the right serves as the reference", "x": 1, "y": 8}, {"code": "# and the code on the right was developed by Sara Amig\"or, while the code on the left serves as the reference", "x": 1, "y": 9}, {"code": "# Additionally, our code on the left developed to handle random data. So some certain sections has been deleted from", "x": 1, "y": 10}, {"code": "# the code on the right", "x": 1, "y": 11}, {"code": "# as well as necessary modification to apply CORVINA, BFGS-B, NELDER-MEAD and SLSQP minimization methods.", "x": 1, "y": 12}, {"code": "#", "x": 1, "y": 13}, {"code": "#", "x": 1, "y": 14}, {"code": "# This file provides useful tools for painting and saving data according to the problem.", "x": 1, "y": 15}, {"code": "# the minimization step, the number of qubits and layers.", "x": 1, "y": 16}, {"code": "# Import os, math, np", "x": 1, "y": 17}, {"code": "# import matplotlib.pyplot as plt", "x": 1, "y": 18}, {"code": "# from matplotlib.cm import cm", "x": 1, "y": 19}, {"code": "# from matplotlib.colors import Normalize", "x": 1, "y": 20}, {"code": "#", "x": 1, "y": 21}, {"code": "# def write_summary(chi, problem, qubits, entanglement, layers, method, name):", "x": 1, "y": 22}, {"code": "#     """", "x": 1, "y": 23}, {"code": "#     This function takes the information of a given problem and saves some text files", "x": 1, "y": 24}, {"code": "#     with this information and the parameters which are solution of the problem", "x": 1, "y": 25}, {"code": "#     INPUT: chi: cost function, to choose between 'fidelity_chi' or 'weighted_fidelity_chi'", "x": 1, "y": 26}, {"code": "#         -chi: cost function, to choose between 'fidelity_chi' or 'weighted_fidelity_chi'", "x": 1, "y": 27}, {"code": "#         -problem: name of the problem, to choose between", "x": 1, "y": 28}, {"code": "#             -'circle', '-3 circles', '-hypersphere', '-tricrown', '-non convex', '-sphere', '-squares', '-wave lines'", "x": 1, "y": 29}, {"code": "#             -qubits: number of qubits, must be an integer", "x": 1, "y": 30}, {"code": "#             -entanglement: whether there is entanglement or not in the Ansatz, just 'y/n'", "x": 1, "y": 31}, {"code": "#             -layers: number of layers, must be an integer, if layers == 1, entanglement is not taken in account", "x": 1, "y": 32}, {"code": "#             -method: minimization method, to choose among ['SGD', 'another valid for function scipy.optimize.minimize']", "x": 1, "y": 33}, {"code": "#             -name: a name we want for our files to be save with", "x": 1, "y": 34}, {"code": "#             -alpha: set of parameters needed for the circuit. Must be an array with shape (qubits, layers, 3)", "x": 1, "y": 35}, {"code": "#             -dim: dimension of data", "x": 1, "y": 36}, {"code": "#             -weight: set of parameters needed for the circuit only if chi == 'weighted_fidelity_chi'. Must be an", "x": 1, "y": 37}, {"code": "#                 array with shape (classes, qubits)", "x": 1, "y": 38}, {"code": "#             -acc_train: accuracy for the training set", "x": 1, "y": 39}, {"code": "#             -acc_test: accuracy for the test set", "x": 1, "y": 40}, {"code": "#             -epochs: number of epochs for 'SGD' method. If there is another method, this input has got no", "x": 1, "y": 41}, {"code": "#                 importance", "x": 1, "y": 42}, {"code": "#             -OUTPUT:", "x": 1, "y": 43}, {"code": "#                 This function has got no outputs, but several files are saved in an appropriate folder. The files are", "x": 1, "y": 44}, {"code": "#                 saved with .txt extension. The first file is saved with .theta.txt", "x": 1, "y": 45}, {"code": "#                 .theta.txt saves the theta parameters as a flat array", "x": 1, "y": 46}, {"code": "#                 .alpha.txt saves the weights as a flat array if they exist", "x": 1, "y": 47}, {"code": "#                 .weights.txt saves the weights as a flat array if they exist", "x": 1, "y": 48}, {"code": "#                 .fidelity.txt saves the fidelity of the circuit", "x": 1, "y": 49}, {"code": "#                 .accuracy.txt saves the accuracy of the training set", "x": 1, "y": 50}, {"code": "#                 .merit.txt saves the merit of the circuit", "x": 1, "y": 51}, {"code": "#                 .summary.txt saves the summary of the problem", "x": 1, "y": 52}, {"code": "#                 .problem.txt saves the problem name", "x": 1, "y": 53}, {"code": "#                 .qubits.txt saves the number of qubits", "x": 1, "y": 54}, {"code": "#                 .entanglement.txt saves the entanglement", "x": 1, "y": 55}, {"code": "#                 .layers.txt saves the number of layers", "x": 1, "y": 56}, {"code": "#                 .method.txt saves the minimization method", "x": 1, "y": 57}, {"code": "#                 .name.txt saves the name of the problem", "x": 1, "y": 58}, {"code": "#                 .chi.txt saves the chi value", "x": 1, "y": 59}, {"code": "#                 .acc_train.txt saves the accuracy for training", "x": 1, "y": 60}, {"code": "#                 .acc_test.txt saves the accuracy for test", "x": 1, "y": 61}, {"code": "#                 .epochs.txt saves the number of epochs", "x": 1, "y": 62}, {"code": "#                 .fidelity.txt saves the fidelity of the circuit", "x": 1, "y": 63}, {"code": "#                 .accuracy.txt saves the accuracy of the training set", "x": 1, "y": 64}, {"code": "#                 .merit.txt saves the merit of the circuit", "x": 1, "y": 65}, {"code": "#                 .summary.txt saves the summary of the problem", "x": 1, "y": 66}, {"code": "#                 .problem.txt saves the problem name", "x": 1, "y": 67}, {"code": "#                 .qubits.txt saves the number of qubits", "x": 1, "y": 68}, {"code": "#                 .entanglement.txt saves the entanglement", "x": 1, "y": 69}, {"code": "#                 .layers.txt saves the number of layers", "x": 1, "y": 70}, {"code": "#                 .method.txt saves the minimization method", "x": 1, "y": 71}, {"code": "#                 .name.txt saves the name of the problem", "x": 1, "y": 72}, {"code": "#                 .chi.txt saves the chi value", "x": 1, "y": 73}, {"code": "#                 .acc_train.txt saves the accuracy for training", "x": 1, "y": 74}, {"code": "#                 .acc_test.txt saves the accuracy for test", "x": 1, "y": 75}, {"code": "#                 .epochs.txt saves the number of epochs", "x": 1, "y": 76}, {"code": "#                 .fidelity.txt saves the fidelity of the circuit", "x": 1, "y": 77}, {"code": "#                 .accuracy.txt saves the accuracy of the training set", "x": 1, "y": 78}, {"code": "#                 .merit.txt saves the merit of the circuit", "x": 1, "y": 79}, {"code": "#                 .summary.txt saves the summary of the problem", "x": 1, "y": 80}, {"code": "#                 .problem.txt saves the problem name", "x": 1, "y": 81}, {"code": "#                 .qubits.txt saves the number of qubits", "x": 1, "y": 82}, {"code": "#                 .entanglement.txt saves the entanglement", "x": 1, "y": 83}, {"code": "#                 .layers.txt saves the number of layers", "x": 1, "y": 84}, {"code": "#                 .method.txt saves the minimization method", "x": 1, "y": 85}, {"code": "#                 .name.txt saves the name of the problem", "x": 1, "y": 86}, {"code": "#                 .chi.txt saves the chi value", "x": 1, "y": 87}, {"code": "#                 .acc_train.txt saves the accuracy for training", "x": 1, "y": 88}, {"code": "#                 .acc_test.txt saves the accuracy for test", "x": 1, "y": 89}, {"code": "#                 .epochs.txt saves the number of epochs", "x": 1, "y": 90}, {"code": "#                 .fidelity.txt saves the fidelity of the circuit", "x": 1, "y": 91}, {"code": "#                 .accuracy.txt saves the accuracy of the training set", "x": 1, "y": 92}, {"code": "#                 .merit.txt saves the merit of the circuit", "x": 1, "y": 93}, {"code": "#                 .summary.txt saves the summary of the problem", "x": 1, "y": 94}, {"code": "#                 .problem.txt saves the problem name", "x": 1, "y": 95}, {"code": "#                 .qubits.txt saves the number of qubits", "x": 1, "y": 96}, {"code": "#                 .entanglement.txt saves the entanglement", "x": 1, "y": 97}, {"code": "#                 .layers.txt saves the number of layers", "x": 1, "y": 98}, {"code": "#                 .method.txt saves the minimization method", "x": 1, "y": 99}, {"code": "#                 .name.txt saves the name of the problem", "x": 1, "y": 100}, {"code": "#                 .chi.txt saves the chi value", "x": 1, "y": 101}, {"code": "#                 .acc_train.txt saves the accuracy for training", "x": 1, "y": 102}, {"code": "#                 .acc_test.txt saves the accuracy for test", "x": 1, "y": 103}, {"code": "#                 .epochs.txt saves the number of epochs", "x": 1, "y": 104}, {"code": "#                 .fidelity.txt saves the fidelity of the circuit", "x": 1, "y": 105}, {"code": "#                 .accuracy.txt saves the accuracy of the training set", "x": 1, "y": 106}, {"code": "#                 .merit.txt saves the merit of the circuit", "x": 1, "y": 107}, {"code": "#                 .summary.txt saves the summary of the problem", "x": 1, "y": 108}, {"code": "#                 .problem.txt saves the problem name", "x": 1, "y": 109}, {"code": "#                 .qubits.txt saves the number of qubits", "x": 1, "y": 110}, {"code": "#                 .entanglement.txt saves the entanglement", "x": 1, "y": 111}, {"code": "#                 .layers.txt saves the number of layers", "x": 1, "y": 112}, {"code": "#                 .method.txt saves the minimization method", "x": 1, "y": 113}, {"code": "#                 .name.txt saves the name of the problem", "x": 1, "y": 114}, {"code": "#                 .chi.txt saves the chi value", "x": 1, "y": 115}, {"code": "#                 .acc_train.txt saves the accuracy for training", "x": 1, "y": 116}, {"code": "#                 .acc_test.txt saves the accuracy for test", "x": 1, "y": 117}, {"code": "#                 .epochs.txt saves the number of epochs", "x": 1, "y": 118}, {"code": "#                 .fidelity.txt saves the fidelity of the circuit", "x": 1, "y": 119}, {"code": "#                 .accuracy.txt saves the accuracy of the training set", "x": 1, "y": 120}, {"code": "#                 .merit.txt saves the merit of the circuit", "x": 1, "y": 121}, {"code": "#                 .summary.txt saves the summary of the problem", "x": 1, "y": 122}, {"code": "#                 .problem.txt saves the problem name", "x": 1, "y": 123}, {"code": "#                 .qubits.txt saves the number of qubits", "x": 1, "y": 124}, {"code": "#                 .entanglement.txt saves the entanglement", "x": 1, "y": 125}, {"code": "#                 .layers.txt saves the number of layers", "x": 1, "y": 126}, {"code": "#                 .method.txt saves the minimization method", "x": 1, "y": 127}, {"code": "#                 .name.txt saves the name of the problem", "x": 1, "y": 128}, {"code": "#                 .chi.txt saves the chi value", "x": 1, "y": 129}, {"code": "#                 .acc_train.txt saves the accuracy for training", "x": 1, "y": 130}, {"code": "#                 .acc_test.txt saves the accuracy for test", "x": 1, "y": 131}, {"code": "#                 .epochs.txt saves the number of epochs", "x": 1, "y": 132}, {"code": "#                 .fidelity.txt saves the fidelity of the circuit", "x": 1, "y": 133}, {"code": "#                 .accuracy.txt saves the accuracy of the training set", "x": 1, "y": 134}, {"code": "#                 .merit.txt saves the merit of the circuit", "x": 1, "y": 135}, {"code": "#                 .summary.txt saves the summary of the problem", "x": 1, "y": 136}, {"code": "#                 .problem.txt saves the problem name", "x": 1, "y": 137}, {"code": "#                 .qubits.txt saves the number of qubits", "x": 1, "y": 138}, {"code": "#                 .entanglement.txt saves the entanglement", "x": 1, "y": 139}, {"code": "#                 .layers.txt saves the number of layers", "x": 1, "y": 140}, {"code": "#                 .method.txt saves the minimization method", "x": 1, "y": 141}, {"code": "#                 .name.txt saves the name of the problem", "x": 1, "y": 142}, {"code": "#                 .chi.txt saves the chi value", "x": 1, "y": 143}, {"code": "#                 .acc_train.txt saves the accuracy for training", "x": 1, "y": 144}, {"code": "#                 .acc_test.txt saves the accuracy for test", "x": 1, "y": 145}, {"code": "#                 .epochs.txt saves the number of epochs", "x": 1, "y": 146}, {"code": "#                 .fidelity.txt saves the fidelity of the circuit", "x": 1, "y": 147}, {"code": "#                 .accuracy.txt saves the accuracy of the training set", "x": 1, "y": 148}, {"code": "#                 .merit.txt saves the merit of the circuit", "x": 1, "y": 149}, {"code": "#                 .summary.txt saves the summary of the problem", "x": 1, "y": 150}, {"code": "#                 .problem.txt saves the problem name", "x": 1, "y": 151}, {"code": "#                 .qubits.txt saves the number of qubits", "x": 1, "y": 152}, {"code": "#                 .entanglement.txt saves the entanglement", "x": 1, "y": 153}, {"code": "#                 .layers.txt saves the number of layers", "x": 1, "y": 154}, {"code": "#                 .method.txt saves the minimization method", "x": 1, "y": 155}, {"code": "#                 .name.txt saves the name of the problem", "x": 1, "y": 156}, {"code": "#                 .chi.txt saves the chi value", "x": 1, "y": 157}, {"code": "#                 .acc_train.txt saves the accuracy for training", "x": 1, "y": 158}, {"code": "#                 .acc_test.txt saves the accuracy for test", "x": 1, "y": 159}, {"code": "#                 .epochs.txt saves the number of epochs", "x": 1, "y": 160}, {"code": "#                 .fidelity.txt saves the fidelity of the circuit", "x": 1, "y": 161}, {"code": "#                 .accuracy.txt saves the accuracy of the training set", "x": 1, "y": 162}, {"code": "#                 .merit.txt saves the merit of the circuit", "x": 1, "y": 163}, {"code": "#                 .summary.txt saves the summary of the problem", "x": 1, "y": 164}, {"code": "#                 .problem.txt saves the problem name", "x": 1, "y": 165}, {"code": "#                 .qubits.txt saves the number of qubits", "x": 1, "y": 166}, {"code": "#                 .entanglement.txt saves the entanglement", "x": 1, "y": 167}, {"code": "#                 .layers.txt saves the number of layers", "x": 1, "y": 168}, {"code": "#                 .method.txt saves the minimization method", "x": 1, "y": 169}, {"code": "#                 .name.txt saves the name of the problem", "x": 1, "y": 170}, {"code": "#                 .chi.txt saves the chi value", "x": 1, "y": 171}, {"code": "#                 .acc_train.txt saves the accuracy for training", "x": 1, "y": 172}, {"code": "#                 .acc_test.txt saves the accuracy for test", "x": 1, "y": 173}, {"code": "#                 .epochs.txt saves the number of epochs", "x": 1, "y": 174}, {"code": "#                 .fidelity.txt saves the fidelity of the circuit", "x": 1, "y": 175}, {"code": "#                 .accuracy.txt saves the accuracy of the training set", "x": 1, "y": 176}, {"code": "#                 .merit.txt saves the merit of the circuit", "x": 1, "y": 177}, {"code": "#                 .summary.txt saves the summary of the problem", "x": 1, "y": 178}, {"code": "#                 .problem.txt saves the problem name", "x": 1, "y": 179}, {"code": "#                 .qubits.txt saves the number of qubits", "x": 1, "y": 180}, {"code": "#                 .entanglement.txt saves the entanglement", "x": 1, "y": 181}, {"code": "#                 .layers.txt saves the number of layers", "x": 1, "y": 182}, {"code": "#                 .method.txt saves the minimization method", "x": 1, "y": 183}, {"code": "#                 .name.txt saves the name of the problem", "x": 1, "y": 184}, {"code": "#                 .chi.txt saves the chi value", "x": 1, "y": 185}, {"code": "#                 .acc_train.txt saves the accuracy for training", "x": 1, "y": 186}, {"code": "#                 .acc_test.txt saves the accuracy for test", "x": 1, "y": 187}, {"code": "#                 .epochs.txt saves the number of epochs", "x": 1, "y": 188}, {"code": "#                 .fidelity.txt saves the fidelity of the circuit", "x": 1, "y": 189}, {"code": "#                 .accuracy.txt saves the accuracy of the training set", "x": 1, "y": 190}, {"code": "#                 .merit.txt saves the merit of the circuit", "x": 1, "y": 191}, {"code": "#                 .summary.txt saves the summary of the problem", "x": 1, "y": 192}, {"code": "#                 .problem.txt saves the problem name", "x": 1, "y": 193}, {"code": "#                 .qubits.txt saves the number of qubits", "x": 1, "y": 194}, {"code": "#                 .entanglement.txt saves the entanglement", "x": 1, "y": 195}, {"code": "#                 .layers.txt saves the number of layers", "x": 1, "y": 196}, {"code": "#                 .method.txt saves the minimization method", "x": 1, "y": 197}, {"code": "#                 .name.txt saves the name of the problem", "x": 1, "y": 198}, {"code": "#                 .chi.txt saves the chi value", "x": 1, "y": 199}, {"code": "#                 .acc_train.txt saves the accuracy for training", "x": 1, "y": 200}, {"code": "#                 .acc_test.txt saves the accuracy for test", "x": 1, "y": 201}, {"code": "#                 .epochs.txt saves the number of epochs", "x": 1, "y": 202}, {"code": "#                 .fidelity.txt saves the fidelity of the circuit", "x": 1, "y": 203}, {"code": "#                 .accuracy.txt saves the accuracy of the training set", "x": 1, "y": 204}, {"code": "#                 .merit.txt saves the merit of the circuit", "x": 1, "y": 205}, {"code": "#                 .summary.txt saves the summary of the problem", "x": 1, "y": 206}, {"code": "#                 .problem.txt saves the problem name", "x": 1, "y": 207}, {"code": "#                 .qubits.txt saves the number of qubits", "x": 1, "y": 208}, {"code": "#                 .entanglement.txt saves the entanglement", "x": 1, "y": 209}, {"code": "#                 .layers.txt saves the number of layers", "x": 1, "y": 210}, {"code": "#                 .method.txt saves the minimization method", "x": 1, "y": 211}, {"code": "#                 .name.txt saves the name of the problem", "x": 1, "y": 212}, {"code": "#                 .chi.txt saves the chi value", "x": 1, "y": 213}, {"code": "#                 .acc_train.txt saves the accuracy for training", "x": 1, "y": 214}, {"code": "#                 .acc_test.txt saves the accuracy for test", "x": 1, "y": 215}, {"code": "#                 .epochs.txt saves the number of epochs", "x": 1, "y": 216}, {"code": "#                 .fidelity.txt saves the fidelity of the circuit", "x": 1, "y": 217}, {"code": "#                 .accuracy.txt saves the accuracy of the training set", "x": 1, "y": 218}, {"code": "#                 .merit.txt saves the merit of the circuit", "x": 1, "y": 219}, {"code": "#                 .summary.txt saves the summary of the problem", "x": 1, "y": 220}, {"code": "#                 .problem.txt saves the problem name", "x": 1, "y": 221}, {"code": "#                 .qubits.txt saves the number of qubits", "x": 1, "y": 222}, {"code": "#                 .entanglement.txt saves the entanglement", "x": 1, "y": 223}, {"code": "#                 .layers.txt saves the number of layers", "x": 1, "y": 224}, {"code": "#                 .method.txt saves the minimization method", "x": 1, "y": 225}, {"code": "#                 .name.txt saves the name of the problem", "x": 1, "y": 226}, {"code": "#                 .chi.txt saves the chi value", "x": 1, "y": 227}, {"code": "#                 .acc_train.txt saves the accuracy for training", "x": 1, "y": 228}, {"code": "#                 .acc_test.txt saves the accuracy for test", "x": 1, "y": 229}, {"code": "#                 .epochs.txt saves the number of epochs", "x": 1, "y": 230}, {"code": "#                 .fidelity.txt saves the fidelity of the circuit", "x": 1, "y": 231}, {"code": "#                 .accuracy.txt saves the accuracy of the training set", "x": 1, "y": 232}, {"code": "#                 .merit.txt saves the merit of the circuit", "x": 1, "y": 233}, {"code": "#                 .summary.txt saves the summary of the problem", "x": 1, "y": 234}, {"code": "#                 .problem.txt saves the problem name", "x": 1, "y": 235}, {"code": "#                 .qubits.txt saves the number of qubits", "x": 1, "y": 236}, {"code": "#                 .entanglement.txt saves the entanglement", "x": 1, "y": 237}, {"code": "#                 .layers.txt saves the number of layers", "x": 1, "y": 238}, {"code": "#                 .method.txt saves the minimization method", "x": 1, "y": 239}, {"code": "#                 .name.txt saves the name of the problem", "x": 1, "y": 240}, {"code": "#                 .chi.txt saves the chi value", "x": 1, "y": 241}, {"code": "#                 .acc_train.txt saves the accuracy for training", "x": 1, "y": 242}, {"code": "#                 .acc_test.txt saves the accuracy for test", "x": 1, "y": 243}, {"code": "#                 .epochs.txt saves the number of epochs", "x": 1, "y": 244}, {"code": "#                 .fidelity.txt saves the fidelity of the circuit", "x": 1, "y": 245}, {"code": "#                 .accuracy.txt saves the accuracy of the training set", "x": 1, "y": 246}, {"code": "#                 .merit.txt saves the merit of the circuit", "x": 1, "y": 247}, {"code": "#                 .summary.txt saves the summary of the problem", "x": 1, "y": 248}, {"code": "#                 .problem.txt saves the problem name", "x": 1, "y": 249}, {"code": "#                 .qubits.txt saves the number of qubits", "x": 1, "y": 250}, {"code": "#                 .entanglement.txt saves the entanglement", "x": 1, "y": 251}, {"code": "#                 .layers.txt saves the number of layers", "x": 1, "y": 252}, {"code": "#                 .method.txt saves the minimization method", "x": 1, "y": 253}, {"code": "#                 .name.txt saves the name of the problem", "x": 1, "y": 254}, {"code": "#                 .chi.txt saves the chi value", "x": 1, "y": 255}, {"code": "#                 .acc_train.txt saves the accuracy for training", "x": 1, "y": 256}, {"code": "#                 .acc_test.txt saves the accuracy for test", "x": 1, "y": 257}, {"code": "#                 .epochs.txt saves the number of epochs", "x": 1, "y": 258}, {"code": "#                 .fidelity.txt saves the fidelity of the circuit", "x": 1, "y": 259}, {"code": "#                 .accuracy.txt saves the accuracy of the training set", "x": 1, "y": 260}, {"code": "#                 .merit.txt saves the merit of the circuit", "x": 1, "y": 261}, {"code": "#                 .summary.txt saves the summary of the problem", "x": 1, "y": 262}, {"code": "#                 .problem.txt saves the problem name", "x": 1, "y": 263}, {"code": "#                 .qubits.txt saves the number of qubits", "x": 1, "y": 264}, {"code": "#                 .entanglement.txt saves the entanglement", "x": 1, "y": 265}, {"code": "#                 .layers.txt saves the number of layers", "x": 1, "y": 266}, {"code": "#                 .method.txt saves the minimization method", "x": 
```

Text Compare

Text Compare

References

- 1 Pérez-Salinas, A., Cervera-Lierta, A., Gil-Fuster, E. & Latorre, J. I. Data re-uploading for a universal quantum classifier. *Quantum* **4**, 226 (2020).
- 2 Schuld, M., Sinayskiy, I. & Petruccione, F. An introduction to quantum machine learning. *Contemporary Physics* **56**, 172-185 (2015).
- 3 Schuld, M., Bocharov, A., Svore, K. M. & Wiebe, N. Circuit-centric quantum classifiers. *Physical Review A* **101**, 032308 (2020).
- 4 LeCun, Y., Bengio, Y. & Hinton, G. Deep learning. *nature* **521**, 436-444 (2015).
- 5 Li, W. & Deng, D.-L. Recent advances for quantum classifiers. *Science China Physics, Mechanics & Astronomy* **65**, 220301 (2022).
- 6 Helstrom, C. W. Quantum detection and estimation theory. *Journal of Statistical Physics* **1**, 231-252 (1969).
- 7 Nielsen, M. A. & Chuang, I. L. *Quantum computation and quantum information*. (Cambridge university press, 2010).
- 8 Hornik, K. Approximation capabilities of multilayer feedforward networks. *Neural networks* **4**, 251-257 (1991).
- 9 Cerezo, M., Verdon, G., Huang, H.-Y., Cincio, L. & Coles, P. J. Challenges and opportunities in quantum machine learning. *Nature Computational Science* **2**, 567-576 (2022).
- 10 Liu, D. C. & Nocedal, J. On the limited memory BFGS method for large scale optimization. *Mathematical programming* **45**, 503-528 (1989).
- 11 Zhu, C., Byrd, R. H., Lu, P. & Nocedal, J. Algorithm 778: L-BFGS-B: Fortran subroutines for large-scale bound-constrained optimization. *ACM Transactions on mathematical software (TOMS)* **23**, 550-560 (1997).
- 12 Liu, Y. *et al.* On centroidal Voronoi tessellation—energy smoothness and fast computation. *ACM Transactions on Graphics (ToG)* **28**, 1-17 (2009).
- 13 Wang, L., Zhou, K., Yu, Y. & Guo, B. Vector solid textures. *ACM Transactions on Graphics (TOG)* **29**, 1-8 (2010).
- 14 Byrd, R. H., Lu, P., Nocedal, J. & Zhu, C. A limited memory algorithm for bound constrained optimization. *SIAM Journal on scientific computing* **16**, 1190-1208 (1995).
- 15 Virtanen, P. *et al.* SciPy 1.0: fundamental algorithms for scientific computing in Python. *Nature methods* **17**, 261-272 (2020).
- 16 Bonet-Monroig, X. *et al.* Performance comparison of optimization methods on variational quantum algorithms. *Physical Review A* **107**, 032407 (2023).
- 17 Pellow-Jarman, A., Sinayskiy, I., Pillay, A. & Petruccione, F. A comparison of various classical optimizers for a variational quantum linear solver. *Quantum Information Processing* **20**, 202 (2021).
- 18 Nelder, J. A. & Mead, R. A simplex method for function minimization. *The computer journal* **7**, 308-313 (1965).
- 19 Gao, F. & Han, L. Implementing the Nelder-Mead simplex algorithm with adaptive parameters. *Computational Optimization and Applications* **51**, 259-277 (2012).
- 20 Abel, S., Blance, A. & Spannowsky, M. Quantum optimization of complex systems with a quantum annealer. *Physical Review A* **106**, 042607 (2022).
- 21 Lockwood, O. An empirical review of optimization techniques for quantum variational circuits. *arXiv preprint arXiv:2202.01389* (2022).
- 22 Kraft, D. A software package for sequential quadratic programming. *Forschungsbericht- Deutsche Forschungs- und Versuchsanstalt für Luft- und Raumfahrt* (1988).